Enhancing Real-time Embedded Image Processing Robustness on Reconfigurable Devices for Critical Applications

POLITECNICO DI TORINO

SCUOLA INTERPOLITECNICA DI DOTTORATO

Doctoral Program in Computer and Control Engineering

Final Dissertation

# Enhancing Real-time Embedded Image Processing Robustness on Reconfigurable Devices for Critical Applications



Pascal TROTTA

Tutor
prof. Paolo PRINETTO

Co-ordinator of the Research Doctorate Course
prof. Matteo SONZA REORDA

04/04/2016

# ACKNOWLEDGEMENTS

This thesis represents the result of three years of work. Throughout this period I had the opportunity to interact with many people that gave me suggestions and encouraged me.

First of all, it is a great pleasure to thank my advisor Prof. Paolo Prinetto, from Politecnico di Torino, that guided me during these three years (and more) with his always precious hints and guidelines. His suggestions led me to grow both from the technical and, most important, personal point of views. I also wish to thank Prof. Stefano Di Carlo, from Politecnico di Torino, that helped me with his valuable experience.

The work that has been carried out during the last three years relied on the collaboration with industrial partners. In particular, I would like to thank Ing. Andrea Martelli, Ing. Antonio Tramutola, and Ing. Piergiorgio Lanza from Thales Alenia Space Italy for our fruitful meetings and their very helpful guidelines. Moreover, during the PhD period I had the opportunity to spend one year in Sweden, working in Cobham Gaisler AB. It is a pleasure to thank the whole Cobham Gaisler team, and in particular Jan Andersson for his immense patience in tutoring me during the entire visiting period, in which I had the opportunity to greatly improve my technical knowledge.

Also, I would like to thank all my colleagues that have worked, or are still working, with me in the laboratory of the Control and Computer Engineering Department at Politecnico di Torino. They made this research period much more enjoyable. In particular, many thanks go to Giulio and Daniele (or Daniele and Giulio) for the great working and non-working time spent together.

Last, but not least, I would like to thank my family that supported and tolerated me during these years.

# CONTENTS

# LIST OF FIGURES

# LIST OF TABLES

CHAPTER 1

# INTRODUCTION

Nowadays, computer vision is one of the most evolving areas of Information Technology (IT). Digital image processing, i.e., the use of algorithms to process and/or extract information from digital images, is being increasingly adopted in multiple application fields.

In general, digital image processing applications fall in two macro-fields: the former includes those applications in which it is used to improve visual information that must be subsequently interpreted by a human actor. Two examples are represented by consumer electronics, such as digital video or photo-cameras, and medical applications. The latter macro-field includes applications in which image processing is employed to extract information that must be stored, transmitted, or used for autonomous machine interpretation. Unmanned Aerial Vehicles (UAVs) navigation, autonomous surveillance and reconnaissance, military situational awareness, and other particular defense, aerospace, and automotive applications, fall in this class [93].

Within the aforementioned fields, image processing is used to serve both non-critical and critical tasks. As example, in automotive, cameras are becoming key sensors for increasing car safety and driving comfort, and for building high-reliable Advanced Driver Assistance Systems (ADAS) [209]. They have been employed for infotainment (non-critical), as well as for some mission-critical driver assistance tasks, such as Forward Collision Warning and Avoidance, Intelligent Speed Control, or Pedestrian Detection.

Also in the aerospace field cameras are likely to become reference sensors. Since decades, cameras are being used for Earth or deep-space observation through in-orbit satellites or powerful space telescopes (e.g., Hubble), and remote rovers navigation. Nonetheless, during the last years, several additional studies and projects carried out by space agencies have also demonstrated suitability of cameras and digital image processing in innovative space-mission contexts. For instance, they can be employed for object recognition during active space debris removal processes [62], or to assist the entry, descent and landing phase of spacecrafts employed in fu-

ture space exploration missions, thus enabling an autonomous video-based navigation of such objects [60].

In these application fields, real-time behaviors are often required in order to allow the system to quickly react to external dangerous events. However, the complexity of the applied algorithms brings a challenge when trying to build real-time embedded image processing systems, requiring high computing capacity, usually not available in modern processors for embedded systems. *Hardware acceleration* is therefore crucial and devices such as Field Programmable Gate Arrays (FPGAs) best fit the growing demand of computational capabilities.

FPGAs are digital integrated circuits which functionality can be programmed, once or multiple times, by the user or the designer after manufacturing, i.e., in the field [100]. They provide logic gates, memories and Digital Signal Processors (DSPs) blocks, thus enabling the implementation of complex digital functions and custom hardware accelerators. Due to their flexibility and continuously growing computing capacity, FPGAs often represent the preferred platform for the final deployment of embedded image processing systems. These devices can assist and offload embedded processors by significantly speeding-up computationally intensive software algorithms, thereby acting as efficient dedicated co-processors.

However, critical applications impose strict requirements in terms of both device dependability and algorithm robustness. Technology shrinking is highlighting reliability problems related to both aging phenomena and to the increasing sensitivity of electronic devices to external radiation events, that can cause transient or even permanent faults, leading to wrong information processed or, in the worst case, to a dangerous system failure.

In addition, even if the circuit which implements the chosen image processing algorithm is working correctly, sensor noise, illumination conditions variation, and other transitory environmental factors can impact the quality of the images acquired by cameras, consequently decreasing the trustworthiness of the algorithm's output results.

The research work presented in this thesis focuses on the development of techniques for implementing efficient and robust real-time embedded image processing hardware accelerators and systems for mission-critical applications. FPGAs have been chosen has target technology, following the current trend that is replacing custom and expensive Application Specific Integrated Circuits (ASICs) with more flexible FPGA devices also in such applications [97].

According to the aforementioned issues, three main challenges have been faced and will be discussed, along with proposed solutions, throughout the thesis: (i) achieving real-time performances, (ii) enhancing algorithm robustness, and (iii) increasing overall system's dependability.

In order to ensure real-time performances, efficient FPGA-based hardware accelerators implementing selected image processing algorithms have been developed. Functionalities offered by the target technology, and algorithm's characteristics have been constantly taken into account while designing such accelerators, in order to efficiently tailor algorithm's operations to available hardware resources. Moreover, efficient design, verification and validation methodologies have

been developed and adopted throughout the research work.

On the other hand, the key idea for increasing image processing algorithms' robustness is to introduce self-adaptivity features at algorithm level, in order to maintain constant, or improve, the quality of results for a wide range of input conditions, that are not always fully predictable at design-time (e.g., noise level variations). This has been accomplished by measuring at run-time some characteristics of the input images, and then tuning the algorithm parameters based on such estimations. Dynamic reconfiguration features of modern reconfigurable FPGAs have been extensively exploited in order to integrate run-time adaptivity into the designed hardware accelerators.

*Dynamic Reconfiguration* is the ability of modern FPGAs to be reconfigured at run-time without interrupting system's operations. The entire device or, in some cases, portions of it can be reconfigured in order to run-time change implemented hardware functionality, to correct design bugs, or to on-line recover from hardware faults.

However, the usage of dynamic reconfiguration exposes the system to new dependability threats that can cause mis-reconfigurations and consequently severely impact reconfigured device's operations. For this reasons, tools and methodologies have been also developed in order to increase the overall system dependability during reconfiguration processes, thus providing safe run-time adaptation mechanisms.

In addition, taking into account the target technology and the environments in which the developed hardware accelerators and systems may be employed, dependability issues have been analyzed and relevant fault models have been defined and adopted. This led to the development of a platform for quickly assessing the reliability and characterizing the behavior of hardware accelerators implemented on reconfigurable FPGAs when they are affected by such faults. The proposed platform can help designers to identify the weaknesses of the circuit and consequently apply the most suitable fault mitigation or protection techniques.

Finally, it is worth to mention that the entire thesis work relies on a strong collaboration with two companies operating in the aerospace market, i.e., *Thales Alenia Space Italy* and *Cobham Gaisler AB* (Sweden). Several target applications in the aerospace field have been selected during the research period and will be considered as main reference case-studies for the proposed solutions. Nonetheless, the concepts behind the contributions of this thesis can be generalized and applied when dealing with the development of embedded image processing, or more in general signal processing, systems for mission-critical applications.

## 1.1 Thesis Organization

The thesis has been split in two main parts in order to allow the reader to easily understand the concepts and contributions of the presented research work. The first part includes two introductory chapters:

- **Chapter 2** provides an overview on image processing concepts and applications. The chapter starts with a brief history of digital image processing. Then, several applications are presented focusing on those ones in which image processing is employed in mission-critical contexts, and highlighting issues, challenges and limitations of modern systems. Several reference use-cases and issues are discussed more in detail since they will be deeply analyzed in the following chapters. Image processing algorithm's classes are introduced and some mathematical background provided;

- **Chapter 3**, provides an overview of modern FPGAs, focusing on reconfigurable ones, and discussing the reasons behind their popularity and adoption in mission-critical applications. After a brief summary of the history that led from the first programmable device to modern reconfigurable FPGAs, the chapter details modern device types and architectures. Dynamic Partial Reconfiguration (DPR) is then introduced along with some technical details about the run-time reconfiguration process. Finally, dependability threats related to the usage of FPGAs in critical environments will be presented, analyzed and discussed.

The second part of the thesis presents the actual contributions of the research work. According to the points and challenges highlighted in the previous section, the rest of the thesis is organized as follows:

- **Chapter 4** provides details on proposed approaches to enhance image processing algorithm robustness [59, 60, 61, 63, 77, 121, 122]. The implementation of associated hardware accelerators and systems on FPGAs is also detailed. Several case-studies are presented. For each of them, algorithms, issues, and advancements with respect to the state of the art are discussed;

- **Chapter 5** discusses the proposed solutions for enhancing dynamic reconfiguration process dependability [58, 65]. In particular, this chapter details the issues related to run-time dynamic reconfiguration and the effect of mis-reconfigurations. Then, it presents two alternative ways to safely enhance reconfiguration process dependability. The former is essentially based on a set of rules to be applied at design-time [58], while the latter relies on the usage of a flexible hardware reconfiguration manager that must be instantiated in the target system [65]. Both approaches can be employed to safely enable self-adaptivity mechanisms in the designed image processing hardware systems without decreasing the dependability levels required by the target applications;

- **Chapter 6** introduces the problem of evaluating system's robustness with respect to target fault models. It provides an overview of the state-of-the-art methods for emulating or injecting faults in FPGA-based systems, highlighting their limitations. Finally, it presents a methodology, along with an associated hardware platform, for emulating the effects of

selected soft errors types on modern FPGAs. The presented methodology provides designers a powerful tool to quickly evaluate the behaviour of the developed systems in harsh environments [64].

CHAPTER 2

# DIGITAL IMAGE PROCESSING FOR MISSION-CRITICAL APPLICATIONS

According to the literature, *Digital Image Processing* represents a sub-field of the more general *Digital Signal Processing*, and refers to the usage of computer algorithms to process and/or extract useful information from digital images [93].

In general, an *image* can be defined as a *"spatial representation of an object or a scene"* [14], represented trough a continuous function:

$$f(x,y), \quad x,y \in \Re \tag{2.1}$$

where $f$ denotes the value of the analog image at the coordinate $(x,y)$.

In contrast to *Analog Image Processing*, which operates on analog representations of the object or scene, *Digital Image Processing* acts on a spatially-sampled and quantized representation of it, i.e., a *digital image*, which can be represented through a discretized function:

$$\bar{f}[x,y], \quad x,y \in \mathbb{N} \tag{2.2}$$

where $f$ represents the quantized intensity value of the digital image at the sampled coordinate $(x,y)$. Equivalently to Equation 2.2, a digital image can be represented through a two-dimensional array:

$$\bar{f}[x,y] = \begin{bmatrix} f[0,0] & f[0,1] & ... & f[0,C-1] \\ f[1,0] & f[1,1] & ... & f[1,C-1] \\ ... & ... & ... & ... \\ f[R-1,0] & f[R-1,1] & ... & f[R-1,C-1] \end{bmatrix} \tag{2.3}$$

composed of $R$ rows and $C$ columns of picture elements, called *pixels*. The value of each *pixel* stores the quantized *intensity* level of that point in the scene. For digital images, pixels store one

or more integer values, represented trough bit arrays, that result from a quantization process carried out by the adopted imaging sensor circuitry. Common bit-widths for the representation of a pixel are 8 or 24 for grey-scale and colors images, respectively [14].

Usually, digital image processing is carried out by applying one or more image processing algorithms to the frames acquired trough an imaging sensor. As shown in Figure 2.1, algorithms are applied sequentially in order to extract high level information that can be easily handled by the user of the image processing system.

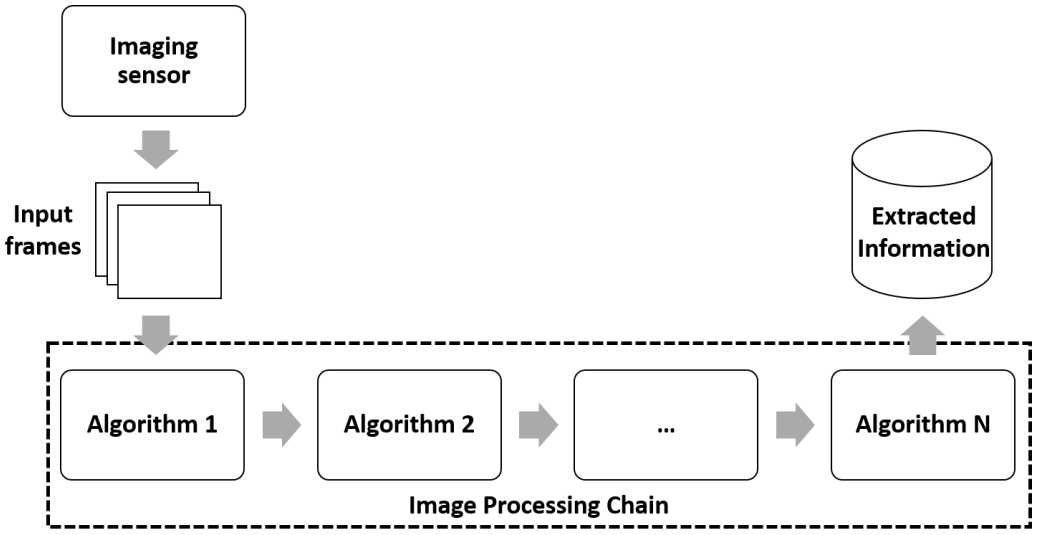



Figure 2.1: Representation of a generic image processing flow.

According to [14], image processing algorithms can be grouped depending on the type of data they have to handle and process. This concept is also illustred in Figure 2.2, which depicts the so called *Image Processing Pyramid*.

- **Image Pre-processing**: this category includes all the algorithms used to enhance the quality of the input frames, and more in general enhance the relevant information. Examples of pre-processing operations are distortion correction, histogram equalization, and noise filtering [93]. Pre-processing techniques act on every pixel composing the image, in order to change their values;

- **Image Segmentation**: the purpose of segmentation is to detect objects or regions in an image, characterized by some specific properties. They essentially extract features from the pixels information, providing a higher-level description of the input frame.

- **Image Classification**: starting from a set of features, classification algorithms identify objects, and eventually classify them into categories;

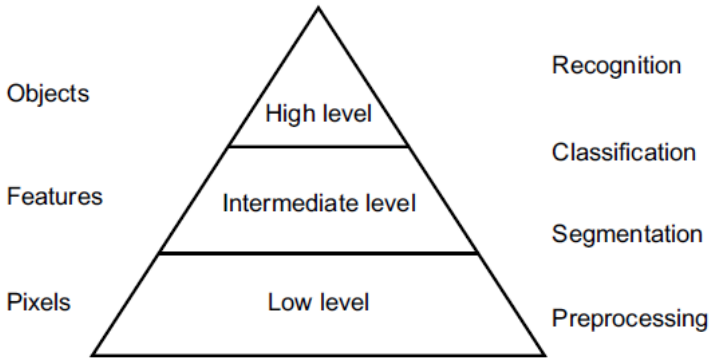



Figure 2.2: Digital image processing pyramid [14].

- **Image Recognition**: recognition algorithms work on objects extracted from the image and aim at deriving high level descriptions or interpretation of the scene.

## 2.1 History of Digital Image Processing

The first applications of digital images date back to the early 1920s, when newspaper pictures were sent through the Bartlane cable transmission system across the Atlantic. At that time, a picture can be trasmitted in less than three hours using a telegraph and reproduced at the receiving part exploiting a special telegraph printer [93, 140].

Although in the following years the systems and underlying technology were improved, we can start talking about *"true"* digital image processing with the introduction and development of digital computers with sufficient computational capabilities. Actually, the first computers powerful enough to process digital images were introduced in the 1960s [93]. In that period, due to the high activity of the space programs, the NASA Jet Propulsion Laboratory developed and used image processing techniques to enhance the quality of the Moon images taken from the *Ranger 7* space probe [158], therefore demonstrating the effectiveness and potentialities of digital image processing. In fact, the imaging system was the unique scientific instrumentation equipped on the *Ranger 7* probe, that in 1964 was able to capture, for the first time (for the US), 4,316 images of the Moon (Figure 2.3) and transmit them to the Earth in the 15 minutes before its impact on the lunar surface [158].

In the late 1960s and early 1970s, digital image processing started to be employed also in medical applications, with the introduction of the computerized axial tomography [87].

Due to the technology advance and the decrease of modern computers cost-to-performance ratio, digital image processing has expanded its application domains. Nowadays it is heavily employed in numerous fields, including, but not limited to, aerospace, automotive, medicine, biology, and defense [93].

Figure 2.3: First image of the Moon taken by *Ranger 7* [158].

## 2.2 Image Processing in mission-critical applications

Nowadays digital image processing is being increasingly adopted in numerous application fields, to serve both critical and non-critical tasks. Two examples are represented by the usage of digital image processing techniques for building innovative automotive Advanced Driver Assistance Systems (ADAS) or autonomous Video-based Navigation Video-based Navigation (VBN) systems for future space exploration missions.

As stated in [7], *"one of the major goals of the automotive industry is to reduce the number of traffic fatalities and severity of accidents"*. Although standard safety (e.g., airbag or ABS) systems are heavily employed in modern vehicles, due to the large and continuously increasing number of vehicles in on the road, new innovative technologies have been introduced to increase car safety and to assist drivers to avoid dangerous situations. In this context, vehicles are being equipped with multiple smart camera-based systems, that can carry out different tasks, as depicted in Figure 2.4.

During the last years, vehicle-based camera systems for ADAS have become more and more complex, being able to perform real-time tasks such as traffic sign recognition, pedestrian detection, forward collision avoidance and road lane detection [167]. An example ADAS system high-level block diagram is illustrated in Figure 2.5.

As reported in [222], an ADAS system may be based on several subsystems that can share some portions of the digital image processing pipeline:

- **Viewable system**: images are captured through one or more imaging sensors and decoded

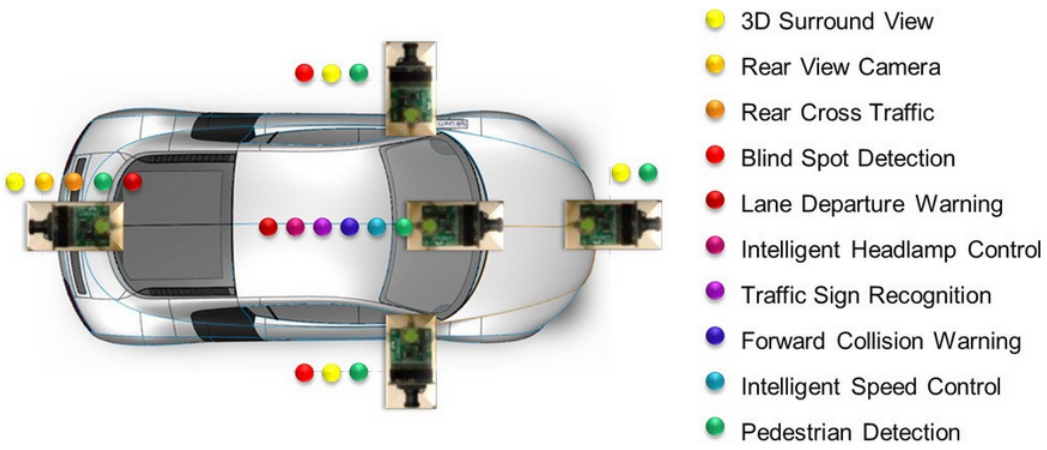



Figure 2.4: Examples of vehicle camera-based systems tasks [222].

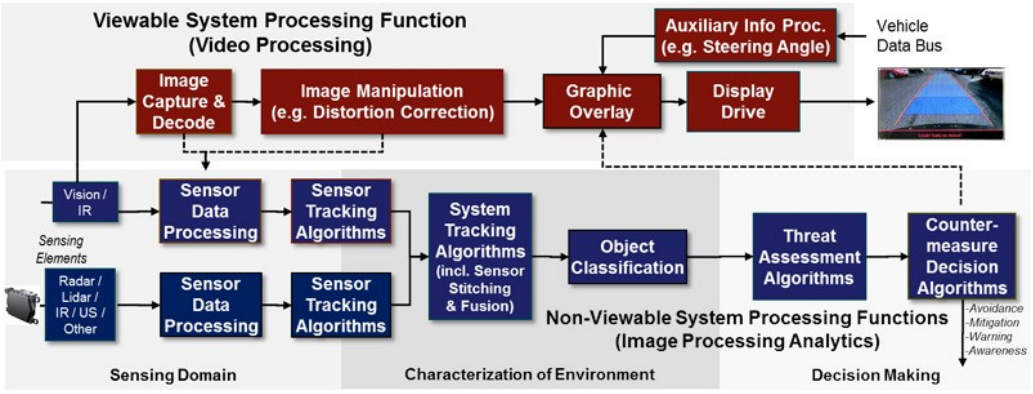



Figure 2.5: Block diagram of an automotive ADAS system [222].

for further video processing (e.g., for image enhancement and distortion correction) in order to provide graphical information that can be displayed to the driver;

- **Non-viewable system**: includes all those hardware and software systems needed to implment digital image processing algorithms for extracting useful information from the acquired frames and for interpreting them to characterize the vehicle environment and support continuous threat assessment and drive countermeasures (e.g., to identifying lane markings, road signs, pedestrians and other vehicles) [222].

On the other hand, in the aerospace domain, digital image processing has been used in the last decades for many purposes [176], such as *Earth Observation* [23, 161], *Space Cartography* [160], *Satellites and Spacecrafts Attitute Control* [129], and *Rovers Navigation* [165].

A recent example of the usage of digital image processing techniques requiring high computational capabilities in the space domain is represented by the *NASA Curiosity* mission [163].

*Curiosity* is a car-sized rover (see Figure 2.6), that is exploring the Martian surface since August 2012.

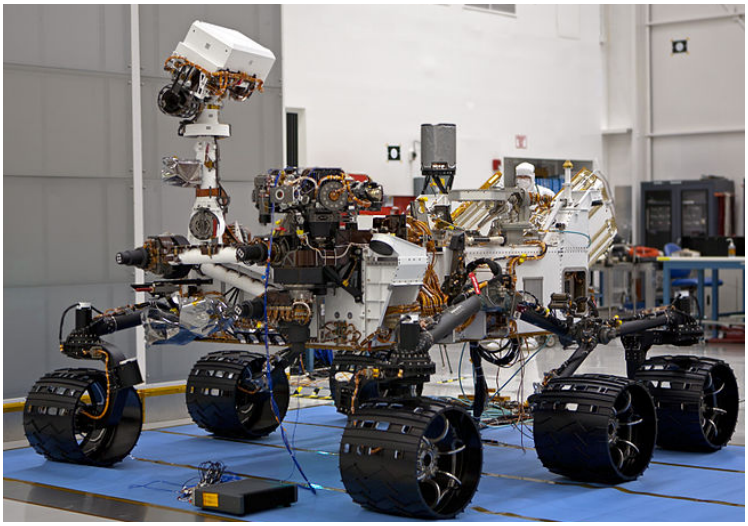


Figure 2.6: *Curiosity* rover.

As reported in [165], the *Curiosity* rover has been equipped with 17 "eyes", for a total of 10 cameras. 6 cameras are devoted to the rover navigation, while the other 4 are used to perform science investigations on the Mars surface:

- **Hazcams**: four pairs of cameras are employed for hazard avoidance. *Hazcams* are black and white cameras mounted on the lower portion of the front and rear of the rover to capture three-dimensional images. Hazard detection is essential to protect the rover from crashing into unexpected obstacles. These cameras work in tandem with the software that allows the rover to autonomously react and avoid dangerous situations [165];

- **Navcams**: two pairs of cameras are used to help rover ground navigation providing a panoramic view of the terrain. These cameras also work in cooperation with *Hazcams*;

- **MastCam**: two *Mast* cameras are used to take three-dimensional color images and videos of the Martian terrain;

- **MAHLI**: the *Mars Hand Lens Imager* provides close views of the minerals, textures, and structures in the martian surface. It takes color images of features as small as 12.5 micrometers.

In addition to the aforementioned cameras, the *Curiosity* mission employs an additional camera that has been used to acquire frames during the rover landing phase. The *Mars Descent Imager* (MARDI) acquired $1600x1200$ pixel images (Figure 2.7) at roughly 5 frames per second throughout the entire landing period, i.e., from the landing module heatshield separation until surface touchdown.

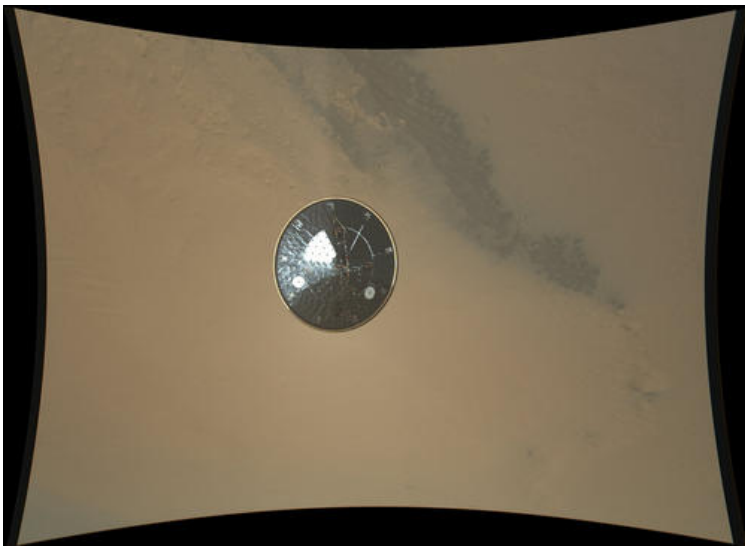


Figure 2.7: Picture of the *Curiosity*'s heatshield taken by *MARDI* [164].

During the landing phase, the acquired frames were written into flash memory in real-time and transmitted later to the Earth.

Although vision-based approaches have been heavily employed to allow autonomous rover ground navigation, during the Entry, Descent and Landing phase, cameras are usually employed as passive components, not included in the spacecraft landing control loop (e.g., *MARDI* in *Curiosity* mission).

In a space exploration mission, the Entry, Descent and Landing (EDL) represents one of the most dangerous phases, since all the expensive on-board electronic and mechanical instruments are subject to strong mechanical vibrations that can damage them. Moreover, a high accuracy of the landing point must be guaranteed in order to be able to reach predefined and safe portions of the target planet surface. For these reasons, during the last years, there was an increasing interest of space agencies in developing innovative EDL systems that try to minimize the probability of mission failures, while at the same time providing an increased precision of the landing point. The landing accuracy is usually measured resorting to the size of the so called *landing ellipse*, which identifies the predicted landing zone. The size of the landing zone is essentially determined by the uncertainty of numerous environmental factors, that can impact the descending

trajectory during the actual landing phase. Figure 2.8 illustrates the predicted landing ellipse of the *Curiosity* rover. The size of the landing ellipse was $20x7$ kilometers.

Figure 2.8: Representation of the *Curiosity*'s landing ellipse on the Martian surface[201].

The high precision of the state-of-the-art *Curiosity*'s landing system is due to the adoption of a combinations of different sensors and approaches that enable a guided landing phase. Figure 2.9 summarizes the *Curiosity*'s EDL phases.

When the parachute is completely deployed, the heat shield is released, and the radar begins collecting data about velocity and altitude. The information gathered from the radar instruments are then used by the Guidance Navigation and Control (GNC) system to correct, if necessary, the spacecraft descending trajectory through the usage of thrusters.

During the last years, space agencies have increased the effort for the development of novel EDL systems. In particular, VBN approaches have been preferred, as demonstrated by several funded studies [69, 79] due to the potentialities offered by digital image processing and the limited weight of vision instruments and associated processing systems. VBN systems aim at computing the speed and attitude of the spacecraft by processing consecutive frames acquired through an imaging sensor. This approach is also called *Relative Navigation*, which aims at computing speed and relative position of the spacecraft by processing images in real-time.

A typical relative navigation image processing chain is depicted in Figure 2.10 [122].

*Relative Navigation* is carried out by performing two main activities:

- **Features Extraction and Matching** FEM: each frame is processed to detect those pixels or regions that represent features of interest in the image (e.g., corners or edges on the surfaces). The detected features are then compared to extract those that can be recognized in two consecutive images (*matching points*);

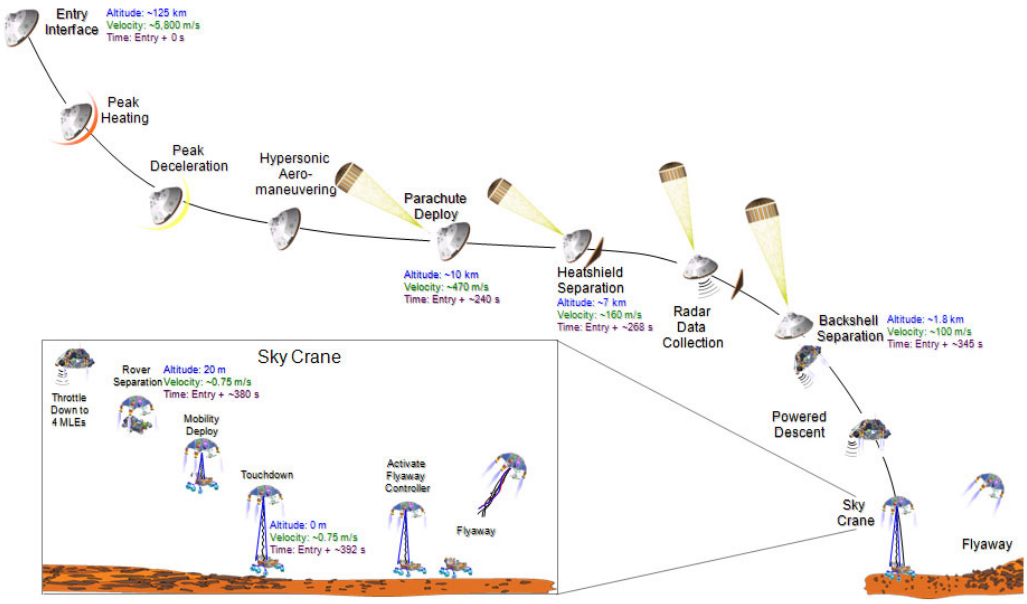



Figure 2.9: Representation of the *Curiosity*'s EDL [159].

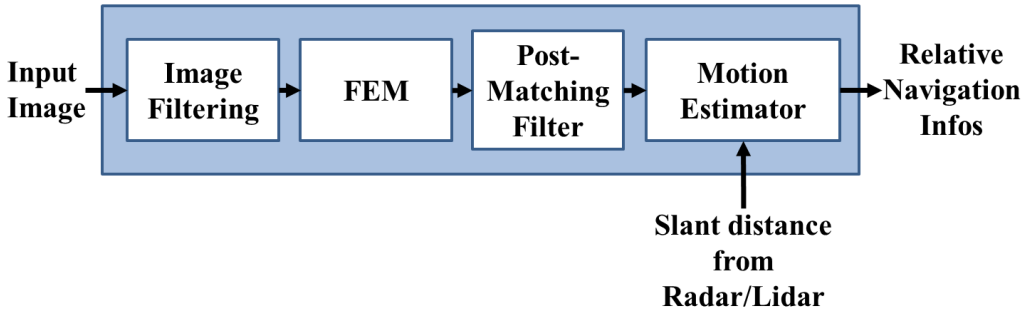



Figure 2.10: Representation of the *Curiosity*'s EDL [122].

- **Motion Estimation**: the output results of *FEM* are analyzed by the motion estimation algorithms that extract the relative position, speed and attitude of the camera by fusing matching points information with those extracted by additional sensor, such as radars or lidars.

A first approach to an autonomous VBN system in space exploration missions has been accomplished during the *Mars Exploration Rovers* mission [162], in which two rovers, i.e., *Spirit* and

*Opportunity* landed on the Martian surface.

The EDL approach adopted in such mission was parachute-assisted and almost entirely ballistic. Moreover, due to the limited knowledge of the Martian environment (e.g., wind speeds and atmosphere composition), the predicted trajectory presented a large uncertainty, leading to an estimated landing ellipse of $15x160$ kilometers.

However, during the *EDL* phase, the so-called Descent Image Motion Estimation System (DIMES) has been employed. DIMES represents the first autonomous machine vision system used to safely land a robotics payload on another planet. It consists of a camera and a software algorithm for estimating horizontal velocity using images, inertial and altitude measurements [40, 109].

Figure 2.11 illustrates the DIMES descent scenario.

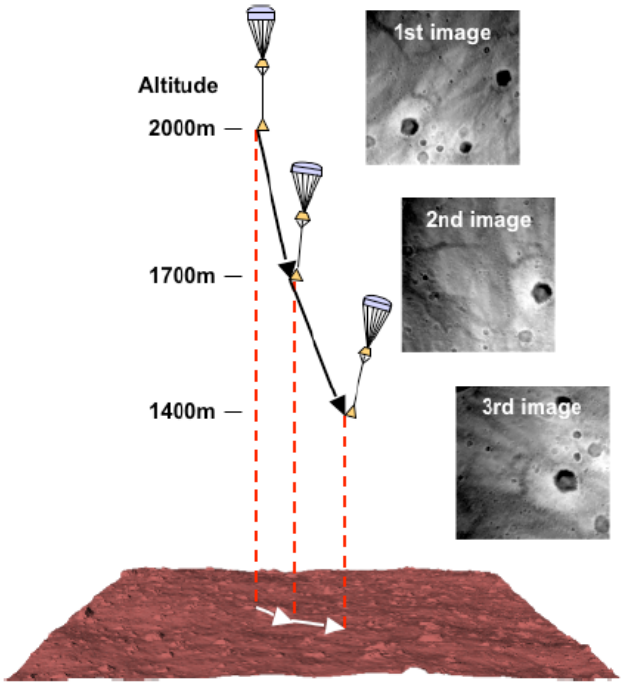



Figure 2.11: *DIMES* descent scenario [109].

Three images are taken by the camera, at roughly 2000m, 1700m, and 1400m above the Martian surface, and processed by a software algorithm that first extracts features between the first and the second image, and other two features between the second and the third image. The three images are scaled and rotated depending on the actual attitude, that has been measured through an inertial measurement unit. Finally, two features for each image pair are tracked using a two-dimensional correlator in order to compute and estimate the average velocity between two acquired frames. The resulting value is also checked with respect to the outputs of the inertial measurement unit data that eventually propagates a control command to thrusters to correct

the horizontal velocity of the spacecraft [40, 109].

It is worth to mention that this system has been successfully employed during landing of both *Spirit* and *Opportunity* rovers, and played an important role in the first case. In fact, during *Spirit* landing, the total velocity was at the limit of the landing airbag capability. Airbags were used to limit the impact forces of the rover with the Martian surface. However, tests highlighted that, if the velocity at impact instant had been too large, the airbags would be ripped [40]. Anyway, DIMES was able to compute the correct values and activate thrusters to reduce the risky horizontal velocity [109] and provide a safe landing of the *Spirit* rover on the planet terrain.

## 2.3 Imaging sensors and related issues

The image sensor represents the fundamental part of a camera. In its basic form, an image sensor can be defined as a sensor that converts light information into electrons, and subsequently electrons into a voltage.

Two main sensor technologies are adopted to generate digital images that can be digitally processed, i.e., Charge-Coupled Device (CCD) and *CMOS*.

Figure 2.12 illustrates the high level block diagram of a CCD sensor.

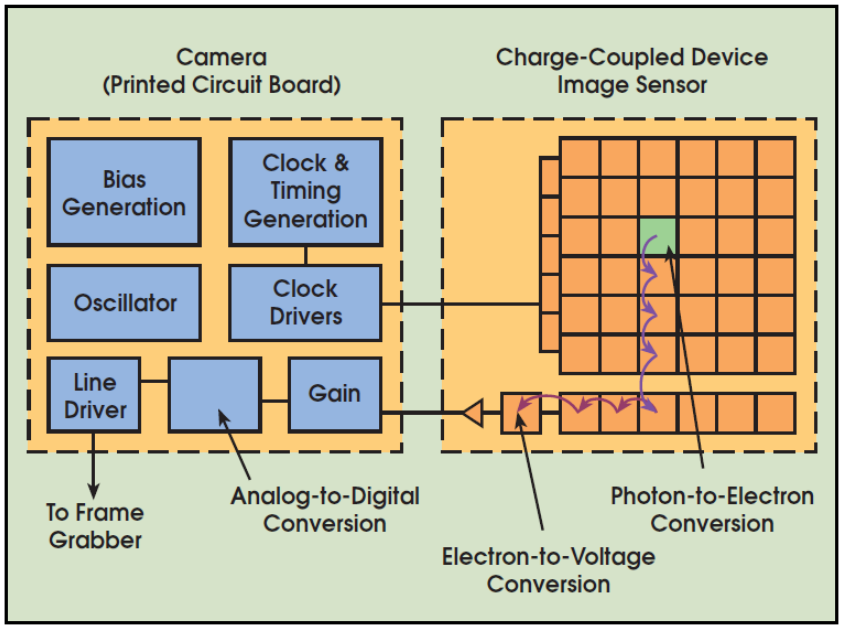



Figure 2.12: CCD imaging sensor high level block diagram [130].

A CCD is a solid-state image sensor composed of a two-dimensional array of pixels. A pixel is a MOS capacitor that stores charges whenever a photon is absorbed [120, 137] (*Photon-to-Electron Convesion* phase). When exposure is complete, pixels' charges are sequentially transferred to the

read-out circuitry that generates a buffered output voltage proportional to the amount of charge trapped by a single pixel. Afterwards, an off-chip Analog-to-Digital Converter (ADC) converts the analog voltage value generated by each pixel into a binary representation.

On the other hand, as depicted in Figure 2.13, in a *CMOS* sensor most of the operations are performed on-chip.

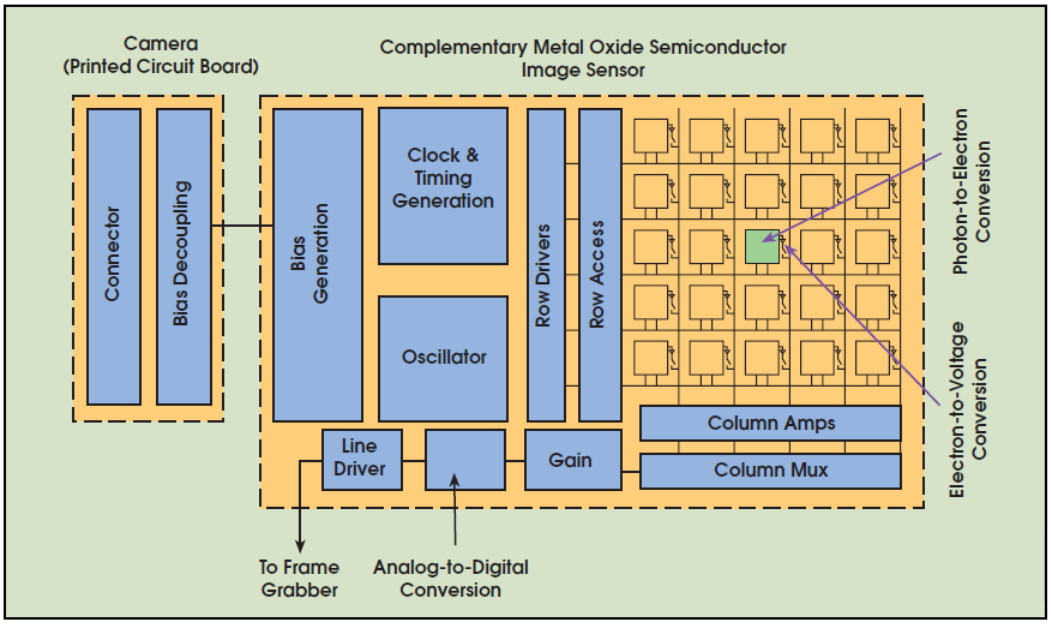



Figure 2.13: *CMOS* imaging sensor high level block diagram [130].

A *CMOS* sensor is composed of a two-dimensional array of pixels elements that, depending on the adopted technology, can be either *passive* (Figure 2.14) or *active* (Figure 2.15).

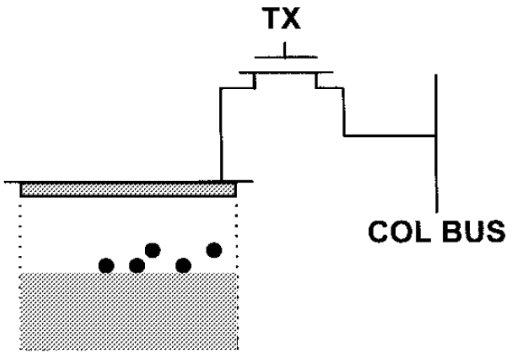



Figure 2.14: *CMOS* passive pixel sensor [80].

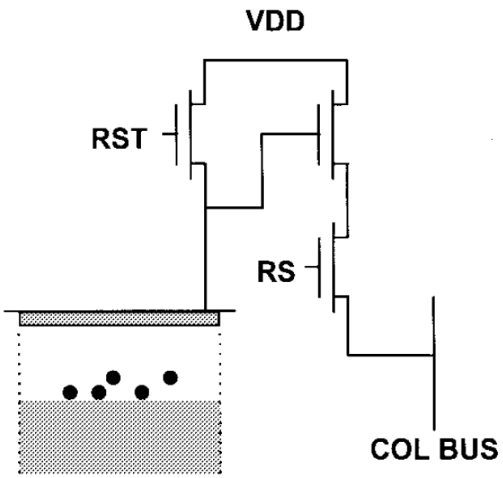



Figure 2.15: *CMOS* active pixel sensor [80].

As shown in Figure 2.14, a *CMOS* passive pixel element consists of a photo-diode and a pass transistor. When the pass transistor is activated, the photo-diode is connected to the *column* bus and the charge accumulated in the photodiode is converted into a proportional voltage by the read-out circuitry [80].

On the other hand, Figure 2.15 illustrates a *CMOS* active pixel, composed of a photo-diode and three transistors [80]. When the *reset* transistor (i.e., the one driven by the *RST* signal) is turned-on, the photo-diode looses the accumulated charge because of the direct connection to the power supply (i.e., *VDD)*. An amplifier transistor is used to read the pixel voltage without removing the actual accumulated charge, while a selection transistor (i.e., the one driven by the *RS* signal) is used guarantee the access of the read-out circuitry to the target pixel voltage. The main difference of such pixel architecture over a CCD sensor relies on the fact that (i) the entire charge-to-voltage conversion takes place in each pixel [130], and (ii) CMOS pixels do not require special manufacturing techniques since they can be fabricated resorting to the same silicon technology process of the surrounding logic.

Either acquired trough a CCD or a *CMOS* sensor, digital images may not accurately reproduce the target scene due to the presence of several phenomena, such as temperature or sensor manufacturing process variations, or sensor mechanical vibrations during the exposure time, that can induce errors in the output pixels values.

In particular, two main effects are usually considered as unwanted sources of digital image degradation, i.e., *noise* and *blur* [93]. Figure 2.16 and Figure 2.17 illustrate two examples of digital images affected by noise and blur, respectively.

Several types of image noise and blur exists, and they are briefly discussed in the sequel.

Noise can be defined as a stochastic undesired variation of the pixels values that leads to a

(a) Original image. (b) Noisy image.

Figure 2.16: Example of an image affected by noise.

(a) Original image. (b) Blurry image.

Figure 2.17: Example of the blur effect.

deviation from the correct representation of the target scene [255].

As stated in [93], *"the principal sources of noise in digital images arise during image acquisition and/or transmission. The performance of imaging sensors is affected by a variety of factors, such as environmental conditions during image acquisition (e.g., light levels and sensor temperature), and by the quality of the sensing element themselves"*. Some examples of noise sources are:

- **Shot noise**: this type of noise, also called *Photon noise*, is caused by the non-deterministic arrival of photons during the image acquisition process. Basically, during the digital image acquisition interval, photons hit the sensor's sensing elements. However, it is not guaranteed that, if we carry-out two independent acquisitions of the same target scene (with the same light conditions and for the same interval), the number of charges acquired by the

sensor elements will be the same, due to the random arrival of photons. Since the charge accumulated is inherently discretized, this leads to a stochastic variation of the pixel value. This effect is highlighted when the image is acquired with very narrow acquisition periods, since the number of accumulated photo-electrons can be very low. Therefore, the impact of this effect can be limited with the adoption of longer exposure periods;

- **Dark current**: also called *Thermal noise*, it represents the noise contribution given by the thermally-generated electrons in the sensor's pixels. It is proportional to the sensor temperature and to the exposure time. As reported in [255], to partially suppress this type of noise, the average dark current for a given acquisition period can be estimated and substracted just before the ADC conversion. Nonetheless, since the thermal noise contribution is also stochastic, even if the average value is removed, the contribution related to the standard deviation is still present [255];

- **Read noise**: it represents the noise generated by the read-out circuitry during the pixel voltage extraction;

- **Amplifier noise**: it is well known that all analog amplifiers introduce noise. In an imaging sensor, the extracted pixel voltage is amplified before the actual *Analog-to-Digital* conversion. This amplification is necessary to extend the range of the voltages generated by the sensor sensing element to be compliant with the adopted ADC;

- **Quantization noise**: it is mainly introduced during the *Analog-to-Digital* conversion process because of the sampling of the input analog voltage and subsequent conversion into a discretized binary representation. It essentially depends on the adopted ADC levels, or equivalently, on the number bits used to represent the pixel value.

On the other hand, according to [28], blur can be classified in:

- **Motion blur**: it is introduced when there is a relative translation, rotation, zoom, or a combination of them, between the camera and the objects in the target scene;

- **De-focus blur**: this kind of image blur is introduced when the target scene presents objects that have different distances with respect to the camera. It mainly depends on the focal length, on the adopted lens, and on the distance between the camera and the objects;

- **Atmospheric blur**: this type of blur mainly affects remote sensing applications (e.g., Earth observation from space satellites). This effects depends on several environmental factors, such as temperature and atmospheric turbulence, that can lead to unpredictable light refraction phenomena.

The aforementioned blur effects are accentuated when the target scene is poorly lighted and so a greater exposure time is required. However, in many situations there is simply not enough light to avoid using a long shutter speed, leading to an acquired images that is inevitably blurry.

### 2.3.1 Mathematical model

The presence of noise or blur effects in images can severely impact the performances of digital image processing and computer vision algorithms, since image details (such as edges) and, more in general, image information, may be partially lost.

Consequently, one of the aims of digital image processing is to try to restore the information affected by unwanted effects, and therefore provide images that can be well processed by the subsequent algorithms in order to extract *"reliable"* information.

A simplified model of a digital image $g[x, y]$, composed of a two-dimensional array of $R$ rows and $C$ columns of pixels, and affected by blur and noise, can be formalized as follows [28]:

$$g[x, y] = \sum_{a=0}^{R-1} \sum_{b=0}^{C-1} h[a, b] f[x - a, y - b] + n[x, y] \tag{2.4}$$

where $f[x, y]$ denotes the ideal image not affected by any unwanted effect, $n[x, y]$ represents the additive noise that corrupt the image, while $h[a, b]$ is the so-called Point Spread Function (PSF).

Equation 2.4 can be re-written as:

$$g[x, y] = h[a, b] * f[x, y] + n[x, y] \tag{2.5}$$

where $*$ denotes the two-dimensional convolution operator.

Equation 2.4 and Equation 2.5 model the effect of blur through a two-dimensional convolution operation between the PSF and $f[x, y]$. Assuming the absence of noise, the PSF represents the transfer function of the imaging system. As stated in [28], *"if the ideal image"*, i.e., $f[x, y]$ in Equation 2.5, *"would consist of a single intensity point or point source, this point would be recorded as a spread-out intensity pattern"*, i.e., $h[x, y]$ in Equation 2.5. Basically, the PSF represents the system response to a point source, and describes how each point source is *"spreaded"* in the output image. From a system theory point of view, assuming that the considered imaging system is linear, the PSF represents the system impulse response.

Despite its simplicity, this model is often adopted in practice because it provides a good approximation level of the real effects, and low-complexity.

In particular, Equation 2.4 assumes that the PSF function is spatially invariant, meaning that each point of the image is blurred in the same way, independently from its location. Moreover, the blur is assumed to be a process that does not absorb or generate energy [28], therefore:

$$\sum_{x=0}^{R-1} \sum_{y=0}^{C-1} h[x, y] = 1 \tag{2.6}$$

On the other hand, the noise affecting the image is usually assumed to be a random variable, that can be described through a probability density function, considered also spatially-invariant and, most important, uncorrelated with respect to the values of the pixels in the image [93]. In practical cases, a *Gaussian* probability density function [93] (characterized by an average value and a variance $\sigma^2$) is often adopted to describe the noise affecting an image (i.e., *Gaussian noise*). The reader may refer to [93] for further details on other common probability density functions used in image processing.

Finally, it is worth to note that, in an ideal case:

$$h[x,y] = \begin{cases} 1 & \text{if } x = y = 0 \\ 0 & \forall x, y \neq 0 \end{cases} \tag{2.7}$$

and

$$n[x,y] = 0, \quad \forall x, y \tag{2.8}$$

therefore Equation 2.4 and Equation 2.5 become:

$$g[x,y] = f[x,y], \quad \forall x \in [0, R-1], y \in [0, C-1] \tag{2.9}$$

CHAPTER 3

# RECONFIGURABLE DEVICES FOR MISSION-CRITICAL APPLICATIONS: ARCHITECTURES AND ISSUES

A Field-Programmable Gate Array (FPGA) is an integrated circuit that can be programmed, once or multiple times, in the field, after manufacturing. Modern FPGAs offer programmable logic blocks, flexible clock generation and interconnection circuitry, memory blocks and embedded hard DSPs. This kind of resources enables the implementation of any digital function. In addition, in order to allow complex Systems-on-Programmable-Chip (SoPC) development, some specific device families also feature special embedded hard blocks, such as entire microprocessors, memory controllers or high-speed transceivers.

Due to their flexibility, limited product development costs, and continuously growing computing capabilities, nowadays FPGA technology represents a feasible and popular alternative solution to Application Specific Integrated Circuits (ASICs), i.e., custom Integrated Circuits (ICs) designed for a specific and fixed purpose.

The reasons behind FPGAs popularity mainly concern the simplified design flow, that consequently impacts on overall project costs and Time-To-Market (TTM). Figure 3.1 illustrates the standard FPGA-based design flow [221].

After deriving the *functional specifications* from the product requirements, designers generate a Register-Transfer Level (RTL) description of the circuit. This task can be accomplished exploiting a Hardware Description Language (HDL), such as VHDL [41] or Verilog [141], or novel High-Level Synthesis (HLS) tools. With the latter approach the RTL description can be automatically generated resorting to C/C++/SystemC/OpenCL high-level functional models [46, 47, 107, 216, 251]. A first *Behavioral Simulation* is then performed in order to verify the generated RTL description against input specifications. Afterwards, the design is synthesized, placed, and routed exploiting FPGA vendor's Computer-Aided Design (CAD) tools. During this process, the RTL description is translated into a set of predefined physical resources (i.e., logic and interconnections)

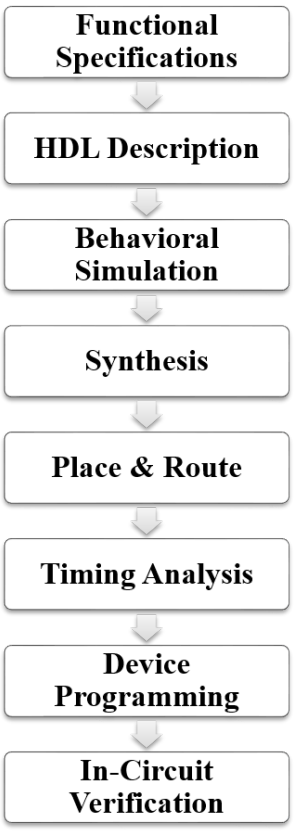



Figure 3.1: Simplified FPGA design flow.

that are then mapped to the ones available in the target FPGA device.

Once the design is fully placed and routed, timing analysis is performed in order to verify if the implemented circuit meets uses-defined timing constraints. This process is straightforward since the physical device is fully pre-characterized by the FPGA manufacturer, and timing models are already embedded in the CAD tool.

Eventually, the device configuration file, namely *bitstream*, is generated. This file stores the physical configuration of each hardware resource inside the FPGA, in order to implement the target functionality. The device is then programmed and can be in-circuit verified. If design bugs are found during this last verification stage, design flow iterations must be performed. Nonetheless, due to FPGA reconfigurability, the same physical device can be programmed with a newly generated and corrected *bitstream*.

On the other hand, the ASICs design flow (Figure 3.2) is much more complex with respect to the FPGA-based one [221].

In particular, since the physical device must be manufactured from the scratch, several addi-

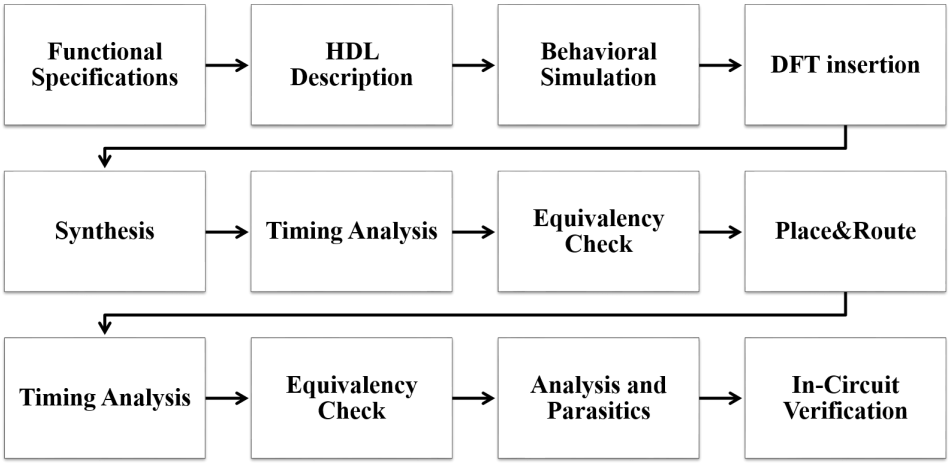



Figure 3.2: Simplified ASIC design flow.

tional verification and characterization steps are required. Various Design-for-Testability (DfT) features, such as scan-chains, Boundary-Scan/JTAG and Built-In Self-Test (BIST), are usually inserted in order to perform device testing using Automated Test Equipment (ATE) [217]. Moreover, *timing analysis* and *equivalency checks* must be also performed after synthesis in order to verify if the synthesis tools have produced a correct result (i.e., *netlist*). Afterwards, all the required information are shared with the silicon foundry that is in charge of generating the masks and manufacturing the physical device. After placement and routing, additional equivalency checks are needed and analysis of second and third order effects, such as noise and parasitics, must be performed.

By comparing Figure 3.1 and Figure 3.2 it is clear that the ASICs design cycle is much more complex, and therefore longer, than the simplified FPGA-based design cycle. The additional verification and analysis steps are mandatory in order to discover as soon as possible design bugs or other issues, with the aim of reducing as much as possible the probability of iterations in the late design steps.

FPGAs also eliminate the complex and time-consuming floorplanning and production stages of the project since the design logic is synthesized in order to be placed onto an already verified, characterized and manufactured FPGA device. Basically, while costs for the application development and architectural design are faced for both FPGA-based and ASIC designs, the Non-Recurrent Engineering (NRE) costs associated to the expensive actual silicon implementation must not be afforded in case of FPGA-based designs, since the physical device production step has been already managed by the FPGA manufacturer.

On the other hand, the choice of an ASIC technology provides full custom design capabilities

that can lead to smaller Integrated Circuit (IC) form factor and higher performances and lower power consumption, since the placement of hardware logic and routing resources can be fully customized and optimized.

However, the selection between one of the two approaches strongly depends also on the production volumes. Figure 3.3 illustrates a qualitative comparison of a project total cost with respect to the product volume for two generic FPGA/ASIC *Technology Nodes*, i.e., *TN1* and *TN2*, where *TN2* is more advanced with respect to *TN1*.

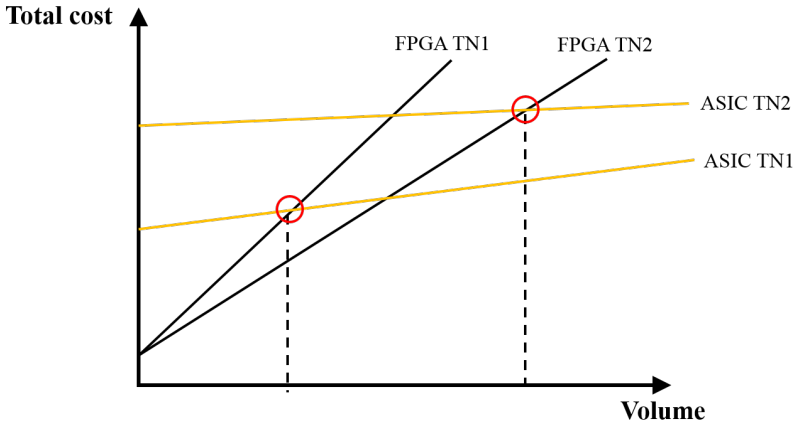



Figure 3.3: FPGAs versus ASICs project cost analysis.

For low volumes, FPGAs are preferred since ASICs high NRE costs, that are increasing with the technology shrinking, may be not affordable. Nonetheless, for high volumes, since the unit cost for an ASIC will be much lower than an FPGA, there will be a crossover point in which producing custom ASICs will be more advantageous. It is worth to note that the crossover volume value increases for each technology advance.

All the aforementioned figures, merged with the reduction of the performance, speed and power gaps between FPGAs and ASICs are pushing designers to adopt FPGAs not only for prototyping purposes [90, 91], but also for the implementation of final release products with low time-to-market in a broad range of applications, especially in those ones requiring medium or low volumes [2, 181]. As a matter of fact, nowadays FPGAs are employed in numerous application domains, including but not limited to, aerospace, defense, automotive, medical, high performance computing, industrial, wireless communication and consumer electronics (see Section 3.3).

## 3.1 History and Evolution of Programmable Logic Devices: from Programmable Logic Arrays to modern FPGAs

Field Programmable Gate Arrays today represent the last and the most advanced evolution of Programmable Logic Devices (PLDs). Depending on their architectural complexity, PLDs can be classified in two main groups (Figure 3.4): (i) Simple Programmable Logic Devices (SPLDs), including Programmable Logic Arrays (PLAs) and Programmable Array Logic (PAL), and (ii) High Capacity Programmable Logic Devices (HCPLDs), which include Complex Programmable Logic Devices (CPLDs) and FPGAs.

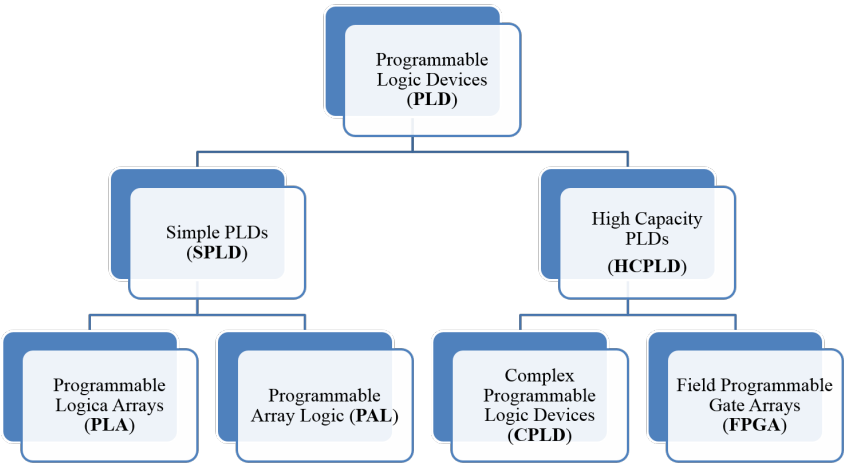



Figure 3.4: Programmable Logic Devices taxonomy.

From the beginning of digital circuits, there has been a desire to have programmable hardware. Consequently, starting from the late 60's, several PLDs architectures were developed and commercialized (Figure 3.5).

Actually, the first PLD was the Programmable Read Only Memory (PROM), which was commercialized in 1969. As shown in Figure 3.6, PROMs are composed of a fixed AND-gates plane and a programmable OR-gates plane. The AND plane generates all the minterms of the inputs, while the OR plane can be programmed to implement the required function. Exploiting this architecture, a PROM can implement any combinational function with limited inputs and outputs.

However, the PROM does not represent an efficient architecture when implementing logic circuits, since it includes a full decoder for its address inputs.

The first evolution of PROMs was represented by the more flexible PLAs. Similarly to PROMs, PLAs are based on an AND plane and an OR plane. However, in this case, both planes are programmable. Due to their full-programmability, PLAs were characterized by very high propagation delays. For this reason, they were not popular, and in the late 1970s they were replaced by

the Programmable Array Logic (PAL) architecture. PALs consist of a programmable AND plane and a fixed OR array. In order to enable the implementation of sequential circuits, registers (i.e., Flip-Flops (FFs)) were also added to the available resources, making PAL very popular.

Nonetheless, as the technology advances and IC density grows, the architectural limits of PAL structures arose. In particular, increasing the number of inputs and outputs of such structures led to very high fan-ins and fan-outs for the AND and OR gates. Consequently, a more efficient solution was the integration of several PAL structures, that can communicate each other through programmable interconnections, in the same chip. These high-density devices were called CPLDs.

As shown in Figure 3.7, CPLDs are essentially made up of logic blocks and *MacroCells* (MC) accessible through I/O blocks and arranged around a programmable interconnection network.

The logic blocks and the MCs include programmable AND-OR planes and additional combinatorial and sequential logic in order to implement multi-level complex functions, while the *Interconnect* block allows communication between logic blocks.

Although CPLDs features made them very popular, several difficulties were encountered while trying to extend CPLDs architecture to allow higher densities and higher logic capacity.

The result was the introduction of the first commercial Field Programmable Gate Array (FPGA) in the 1985 by *Xilinx* co-founders Ross Freeman and Bernard Vonderschmitt.

In contrast to CPLDs, FPGAs are composed of a two-dimensional array of configurable logic blocks. (Figure 3.8).

Usually, each logic block contains one or more Look-Up Tables (LUTs), several Flip-Flops (FFs), multiplexers, and dedicated carry-chains. This resources allow the implementation of any combinatorial or sequential function. The array of logic blocks is interleaved with a fexi-

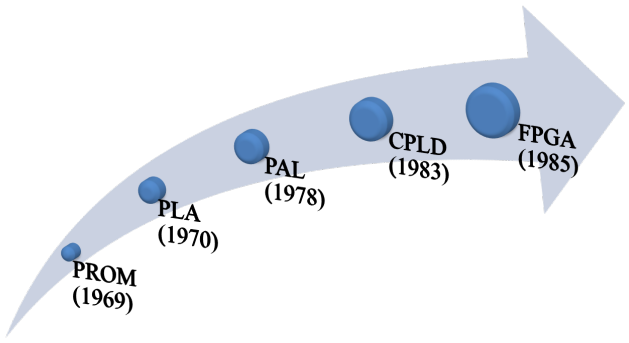



Figure 3.5: Programmable Logic Devices history roadmap.

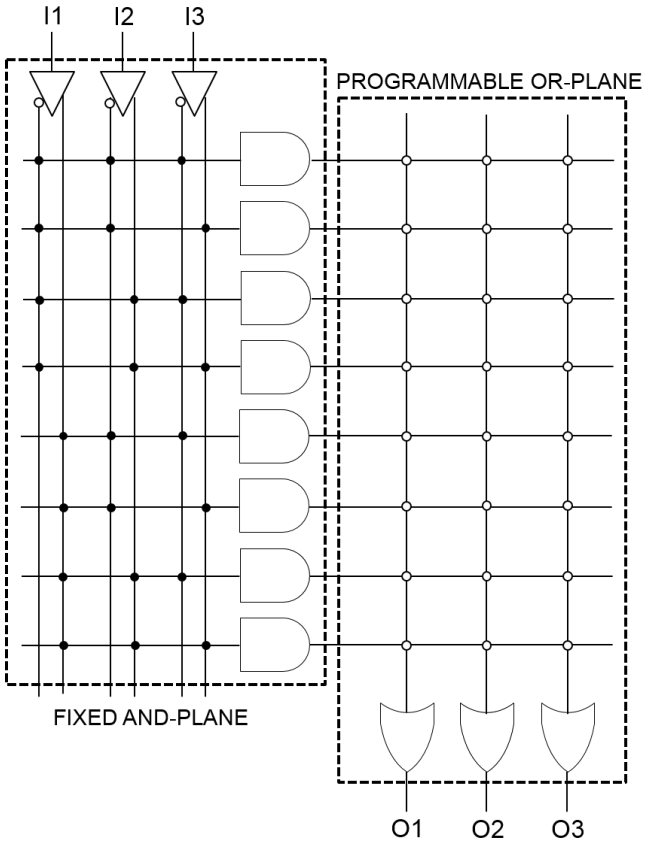



Figure 3.6: PROM internal architecture.

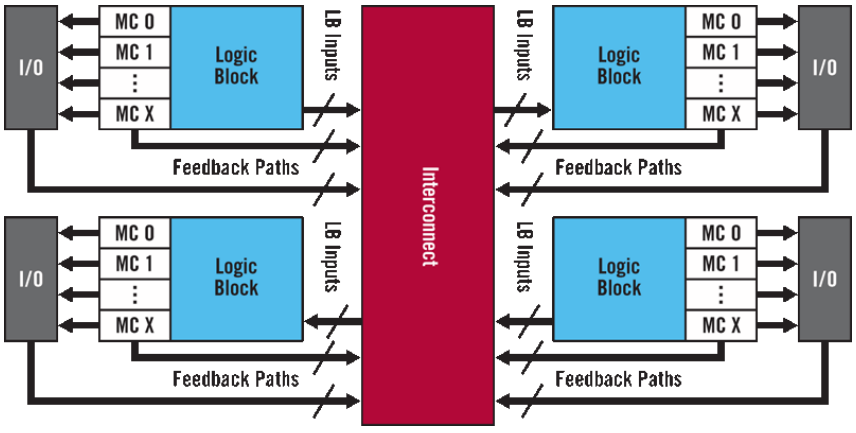



Figure 3.7: CPLD internal architecture [220].

ble interconnect network, configurable through programmable interconnection matrices. Moreover, input/output signals are managed through dedicated I/O blocks, usually supporting multi-standard voltage levels [244]. Modern FPGA devices include also embedded SRAM memory blocks and dedicated DSPs slices. The memory blocks provide a total storage space up to several tens of MegaBytes on high-end devices and are used to implement fast and large data structures. The DSPs slices usually include dedicated and configurable binary signed or unsigned multiply-and-accumulate (MAC) units. The latter are massively used to implement arithmetic functions like parallel multipliers [6].

During the last years, the constant technology shrinking enabled manufacturing of even more heterogeneous FPGAs with increased logic capacity and the integration of entire embedded microprocessors, communication and memory controllers, or high-speed serial transceivers. Recent examples are the *SmartFusion2* [146] and *Zynq-7000* [245] (see Figure 3.9) devices produced by *Microsemi* and *Xilinx*, respectively. [146, 245]. Today, FPGAs available in the market enable the implementation of complex SoPC, consequently reducing device count and board space.

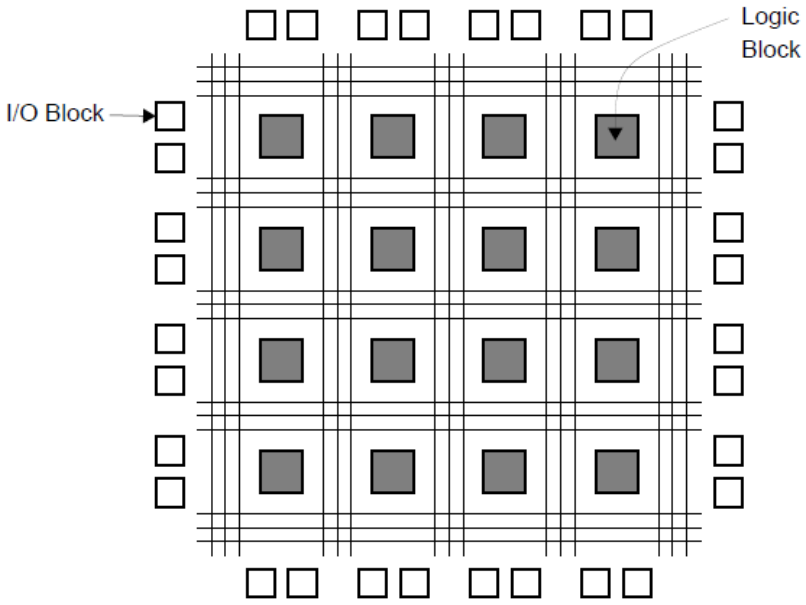



Figure 3.8: FPGA internal architecture [30].

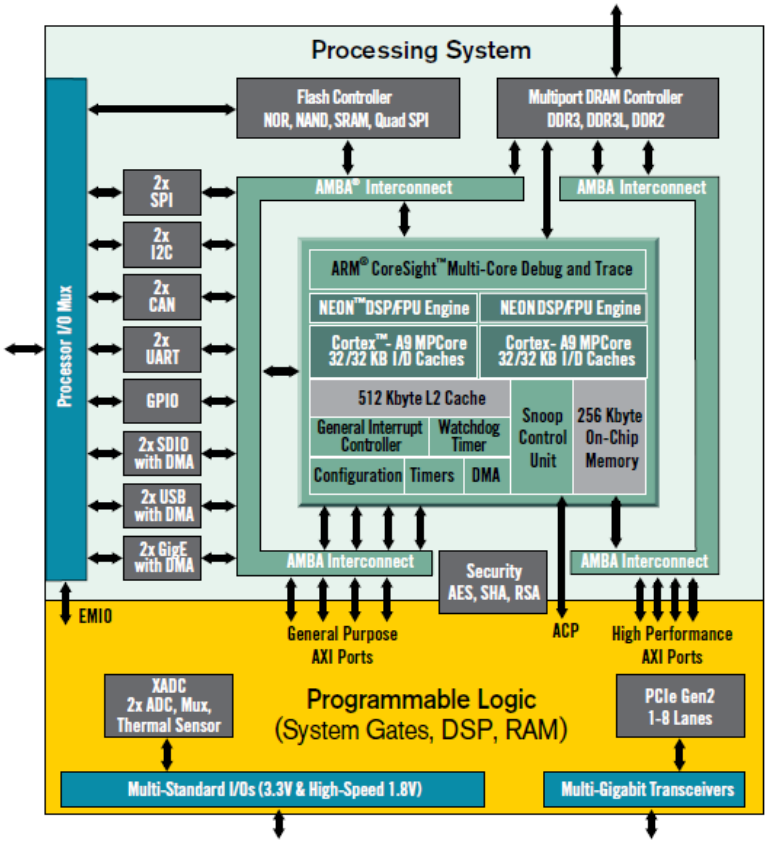



Figure 3.9: Zynq-7000 All Programmable SoC architecture [245].

## 3.2 Field Programmable Gate Arrays architectures

As aforementioned, FPGAs include a set of configurable resources, i.e., logic and interconnects, that can be programmed in order to implement a user-defined logic function. The configuration process essentially consists in downloading the so-called *bitstream* into the target device. The *bitstream* is generated by FPGAs vendors' CAD tools at the end of the design process (see Figure 3.1).

Configuration is stored inside the device exploiting a defined programming technology. Modern FPGAs programming methods basically rely on three different technologies that make them one-time programmable or reconfigurable (Figure 3.10):

- *Antifuse*;
- *Flash memory*;
- SRAM.

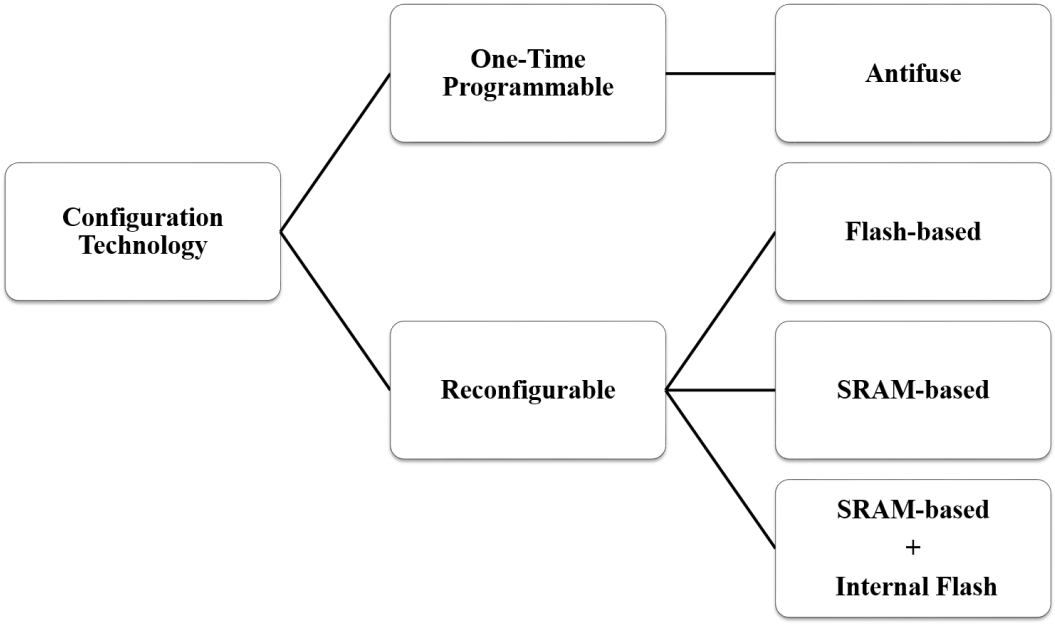



Figure 3.10: FPGAs taxonomy.

The following subsections details these three programming technologies, highlighting advantages and drawbacks.

### 3.2.1 One-time programmable FPGAs

One-time programmable FPGAs use the so-called *Antifuse* technology in order to be configured.

The *Antifuse* is a programmable switch that initially acts as an open-circuit and can be burned in order to provide a very low resistance connection. The burning process is irreversible, hence the *Antifuse*-based FPGAs can be programmed only once. Consequently, the configuration is non-volatile.

Figure 3.11 illustrates the working principle of the *Antifuse* technology.

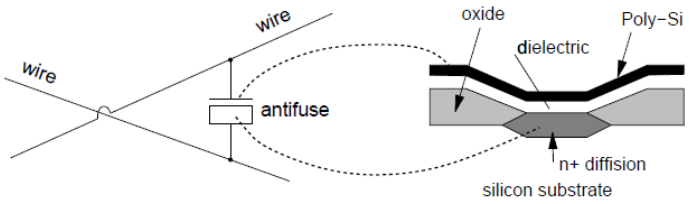



Figure 3.11: Antifuse example [30].

Basically, the *Antifuse* consists of a dieletric layer placed between two conductive layers. In order to build a programmable connection, this device can be placed across two interconnect wires. If unprogrammed, the *Antifuse* isolates the two wires. Applying a high voltage breaks down the dieletric, therefore allowing FPGA signals to pass through the *Antifuse* [119] [139].

Examples of *Antifuse*-based FPGAs are the *Axcelerator* [143] and *RTAX-S/SL* [148] families manufactured by *Microsemi*.

Figure 3.12 shows the architecture and layout of an *Antifuse-based Microsemi RTAX* device.

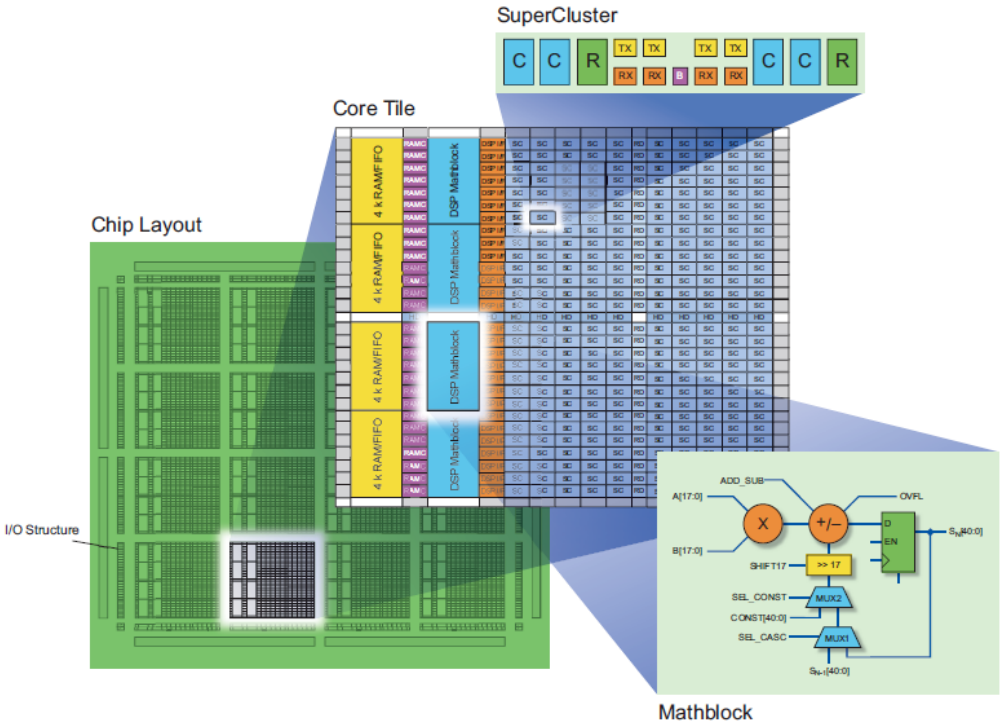



Figure 3.12: *Microsemi RTAX-DSP* device architecture [148].

The programmable logic is organized as a two-dimensional array of *SuperClusters* (SC), and columns of *RAM/First-In First-Out (*FIFO*)* blocks and *DSP Mathblocks*. Each *SuperCluster* features configurable combinational blocks (*C*), registers (*R*) and interconnection buffers (*TX*, *RX* and *B*) that allow the implementation of complex combinatorial or sequential functions. On the other hand, *RAM*/FIFO blocks and *DSP Mathblocks* can be used to implement memory structures and arithmetic operations [148].

### 3.2.2 Reconfigurable FPGAs

Contrary to *One-Time Programmable* FPGAs, *reconfigurable* FPGAs can be re-programmed multiple times and today represent the most dominant part of the FPGA market.

In this case, programmable switches are configured resorting to memory cells used to store the FPGA configuration. Depending on the underlying programming memory technology (i.e., *Flash* or *SRAM*), the configuration can be volatile or non-volatile.

#### 3.2.2.1 Flash-based FPGAs

*Flash-based* FPGAs resort to Flash memory cells to store the device configuration and drive programmable resources accordingly.

Flash memory cells are based on the *floating-gate avalanche-injection MOS* (FAMOS) technology. As shown in Figure 3.13, the standard Metal-Oxide-Semiconductor (MOS) transistor is enriched with an additional *Floating Gate*, isolated from the main *Control Gate*.

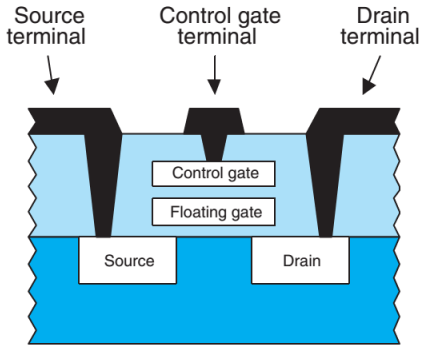



Figure 3.13: Floating Gate transistor [139].

If unprogrammed, the floating gate does not affect the normal operations. When a high voltage is applied between the control gate and drain terminals, negative charges flow through the isolating layer and remain trapped in the floating gate. When the programming signal is removed, a stable negative charge will be present into the floating gate, therefore modifying transistor's behavior. The process can be reversed by applying proper high voltages in order to remove the trapped charges from the floating gate [139].

Figure 3.14 illustrates the usage of such technology in an FPGAs device. Basically, the *Programming Transistor* is used to inject charges into the floating gate, while the *Switching Transistor* represents the actual programmable switch, where FPGA user signal can pass through [119]. By having separate switching and programming transistors, the switching one is preserved from high programming voltages and can interface directly with low voltage logic [184].

The main advantage of Flash-based FPGAs is their non-volatility, i.e., the configuration will not be lost even if the power supply is removed. This eliminates the need for an external storage

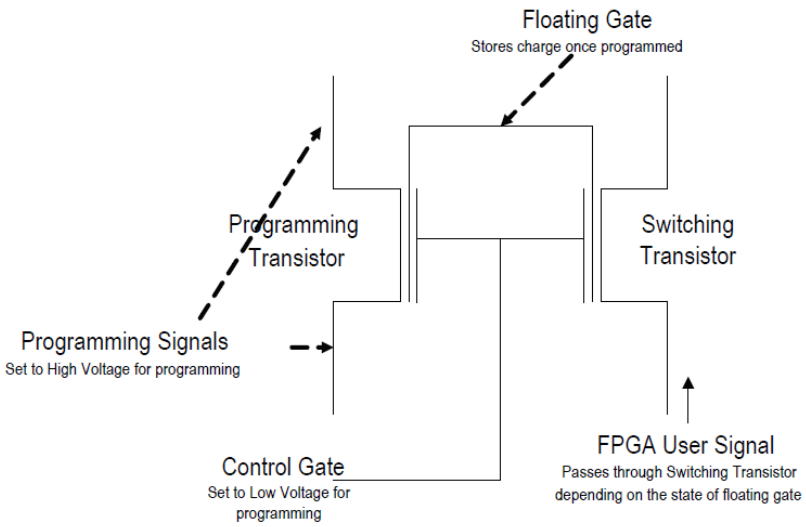



Figure 3.14: FLASH-based FPGAs programming technology [119].

device used to load the configuration at startup. Moreover, with respect to *Antifuse-based* FPGAs, *Flash-based* ones are reprogrammable. Nonetheless, the reconfiguration cycles are limited and dictated by the accuracy of the programming/erasing procedures that can lead to charge biases in the floating gates [119].

Recent examples of *Flash-based* devices are represented by the *SmartFusion2*, *IGLOO2*, and *RTG4* FPGA families, manufactured by *Microsemi* [146, 149].

Figure 3.15 illustrates the layout of a *Microsemi RTG4 FPGA* device.

The device presents a two-dimensional array of *Logic Clusters*, made up of *Logic Elements* including Look-Up Tables (LUTs) and FFs. Rows of embedded memory blocks, i.e., *LSRAMs*, *uSRAM* and *uPROM*, and *DSP Mathblocks* are also interleaved in the *Logic Clusters* array. The device also features embedded Serializer/Deserializer (SERDES) blocks that facilitate the implementation of high speed communication interfaces [149].

#### 3.2.2.2 SRAM-based FPGAs

*SRAM-based* FPGAs exploit static memory cells (Figure 3.16) to store configuration data for programmable switches and logic.

Figure 3.17 illustrates the simplified architecture of an *SRAM-based* FPGA.

In paricular, SRAM cells are used to drive the gate of pass transistors and the select lines of multiplexers in order to build the programmable network that connect and configure logic cells.

Unlike *Flash-based* and *Antifuse-based* devices, *SRAM-based* programming technology offers an infinite reprogrammability and it does not require non-standard technological processes. This

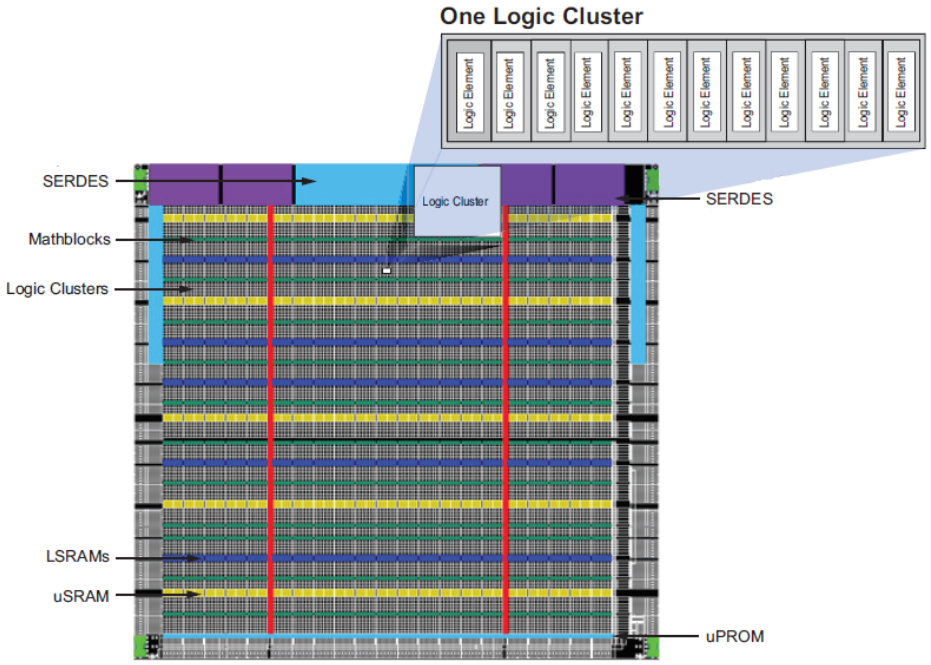



Figure 3.15: *Microsemi RTG4* device architecture [149].

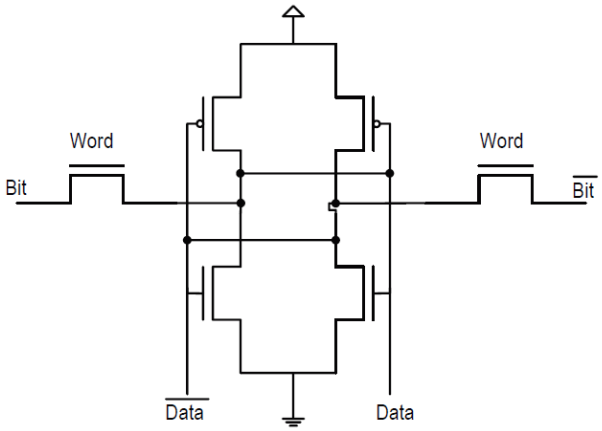



Figure 3.16: SRAM memory cell [119].

greatly simplifies the manufacturing process, since the same technology can be used for both configuration memory and programmable logic. Moreover, due to the adopted underlying technology, programming cycles are faster with respect to *Flash-based* and *Antifuse-based* devices [17].

On the other hand, the main drawback of such programming technology is represented by

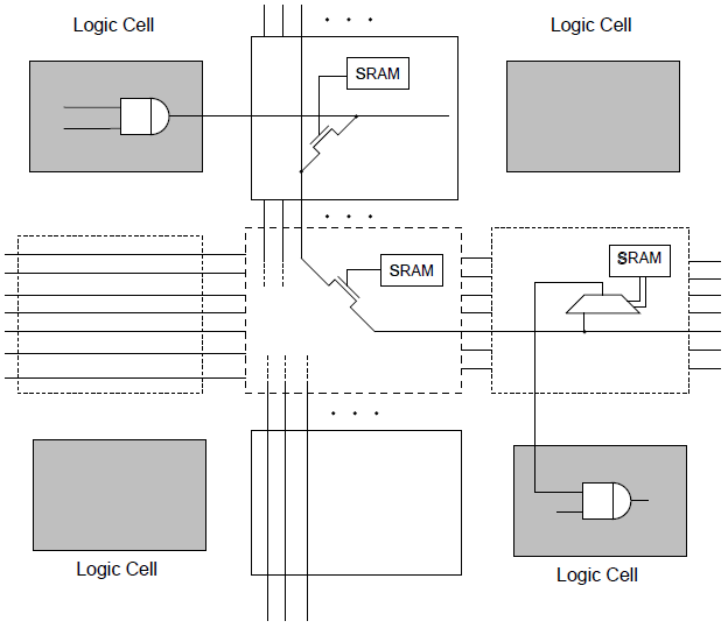



Figure 3.17: SRAM-based FPGAs programming technology [30].

SRAM volatility. *SRAM-based* FPGAs must be programmed at power-on and lose their configuration when the power source is disconnected. Consequently, an additional external storage or control device is required in order to load the FPGA configuration at start-up. This also introduces *security* issues, since configuration information can be intercepted and stolen. However, to counteract such problem, modern *SRAM-based* FPGAs offer configuration encryption services [119]. Some *SRAM-based* device families also offer internal Flash memory blocks to store one or more configuration *bitstreams* that can be automatically loaded at startup in SRAM configuration cells [124, 249].

Popular examples of *SRAM-based* devices are the *Stratix* [8] and *7Series* [244] families manufactured by the two major FPGA vendors, i.e., *Altera* and *Xilinx*, respectively.

The basic architecture of state-of-the-art *Altera Stratix 10* devices is illustrated in Figure 3.18.

The main building block of *Altera*FPGAs is the Adaptive Logic Module (ALM), which includes registers, full adders and a LUT (Figure 3.19).

ALMs can be connected together exploiting a programmable interconnection network, that features bypassable registers used for critical paths retiming and pipeling.

Figure 3.20 illustrates the overall block diagram of a *Stratix 10* device, which features the aforementioned core logic fabric architecture along with an ARM-based hard processor system and memory controllers, transceivers, DSPs, and memory blocks embedded in the same chip [9].

A similar architecture is also adopted by *Xilinx*. The basic logic building block of a *Xilinx*

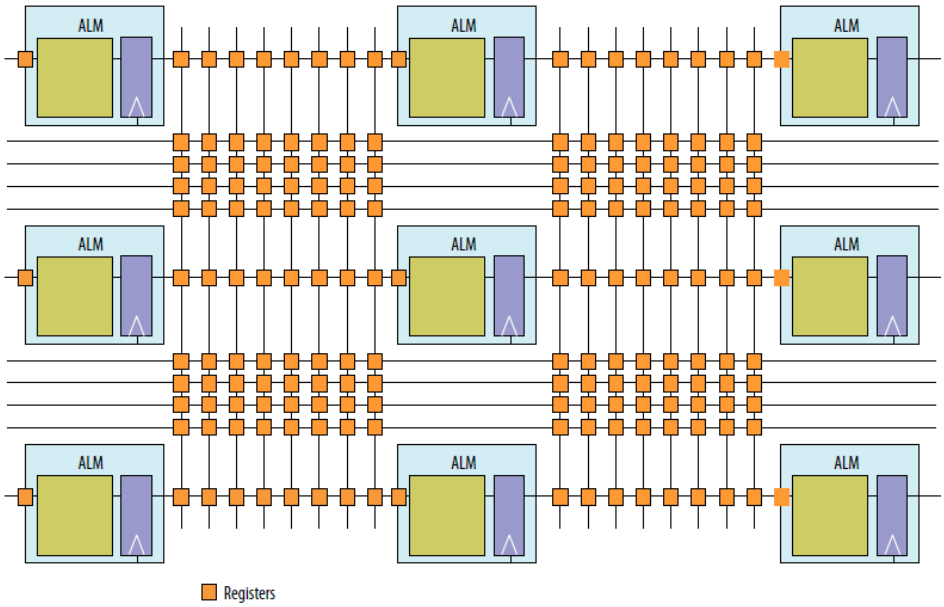



Figure 3.18: Core Logic Fabric for *Altera Stratix 10* devices [9].

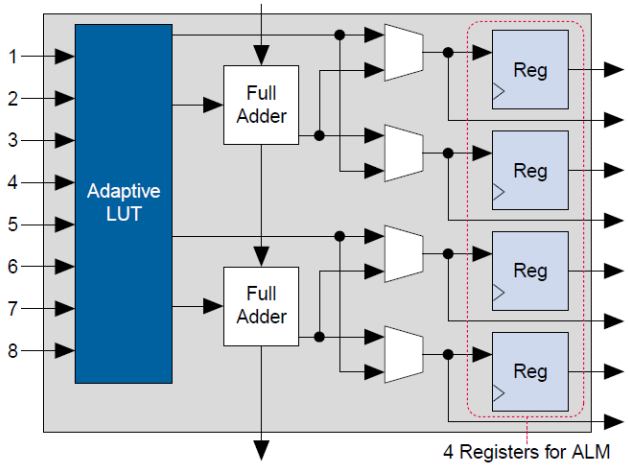



Figure 3.19: *Altera Stratix 10 FPGA ALM Block Diagram* [9].

FPGA is represented by the CLB, that includes logic slices (Figure 3.21). Each logic slices is made up of several LUTs, multiplexers and FFs. Each CLB is also connected to neighbor CLBs by means of dedicated and fast carry lines (i.e., *CIN* and *COUT* in Figure 3.21) and local interconnections, while a configurable *Switch Matrix* provides access to the general routing network.

As can be seen in Figure 3.22, CLBs are arranged as a two-dimensional array interleaved with columns of embedded DSPs and memory blocks.

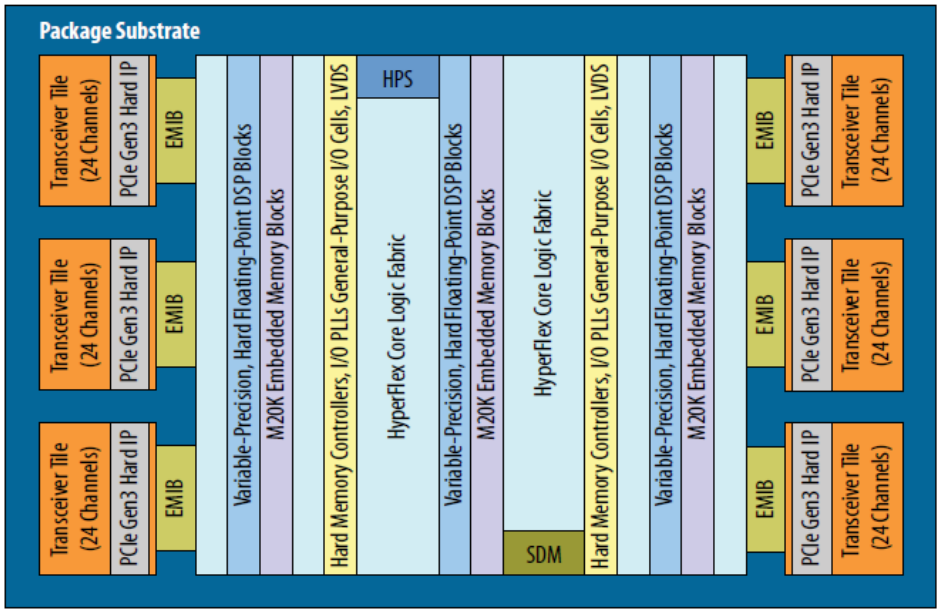



Figure 3.20: *Altera Stratix 10 FPGA* Architecture Block Diagram [9].

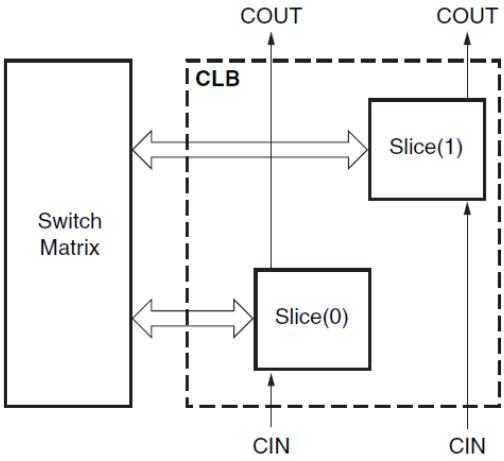



Figure 3.21: *Xilinx* FPGAs CLB architecture[242].

## 3.3 FPGAs for mission-critical applications

Nowadays, FPGAs applications cover numerous fields, including but not limited to, aerospace, defense, automotive, medical, industrial, and wireless communication [104, 225]. FPGAs appli-

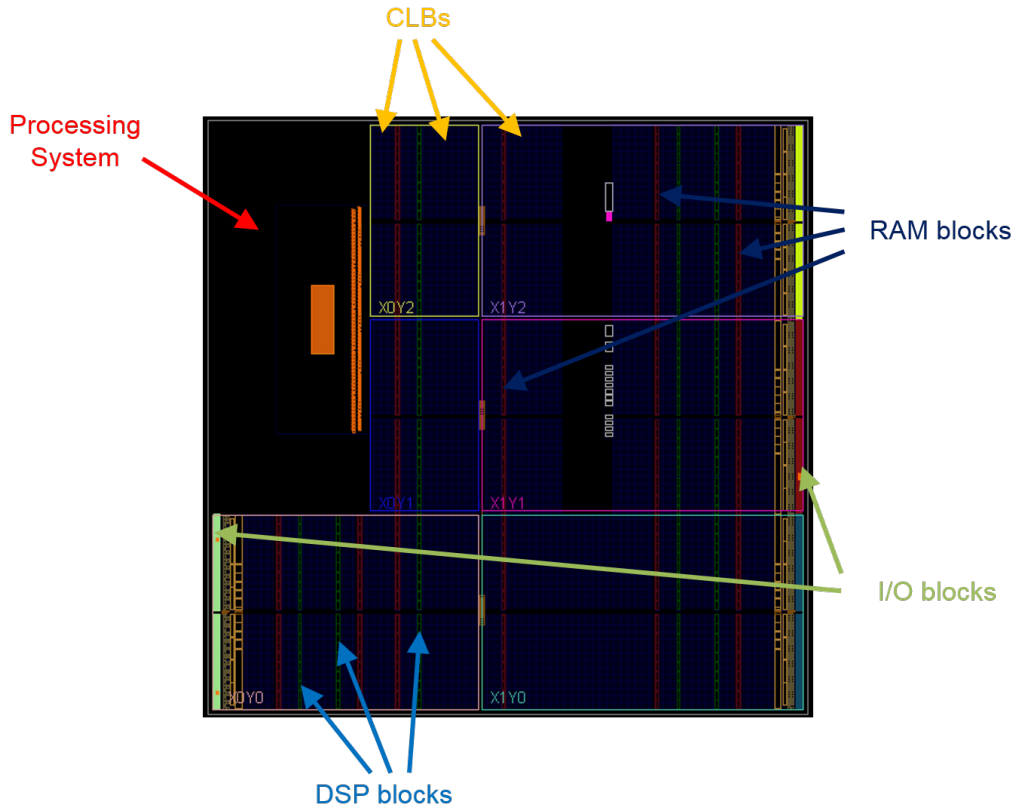



Figure 3.22: Layout of a *Xilinx Zynq-7000* FPGA.

cations can span from high-performance computing data centers [173, 208] to digital signal and image processing in real-time embedded systems [142, 203].

Due their characteristics, FPGAs are also widely employed in safety- and mission-critical scenarios, and in general, in those applications demanding high dependability [17], such as in space where, as example, the development time and costs for producing new ASICs are not affordable due to the very narrow market.

As a matter of fact, *Antifuse*, *Flash* and *SRAM*-based FPGAs have been widely adopted in the space industry. During the last decade, both *Microsemi* and *Xilinx* produced space-grade devices that meet the technology requirements imposed by the harsh environment in which they must operate.

*Microsemi* provides radiation-tolerant *Antifuse* and *Flash-based* FPGAs, such as the *RTSX-SU*, *RTAX*, and *RT ProASIC3* device families [147]. Figures 3.23-3.24-3.25-3.26 show a list of space missions in which these devices have been adopted, or it is planned to adopt them [144].

On the other hand, *Xilinx* provides both radiation-tolerant and radiation-hardened *SRAM*-


Figure 3.23: Space missions employing *Microsemi RTSX-SU* FPGAs [144].


Figure 3.24: Space missions employing *Microsemi RTAX* FPGAs [144].

*based* devices, such as the *Virtex-4QV* or *Virtex-5QV* families [247, 248]. These devices have been also widely employed in the space market.


Figure 3.25: Space missions employing *Microsemi RTAX* FPGAs [144].


Figure 3.26: Past and planned Space missions employing *Microsemi RTAX* FPGAs [144].

FPGAs are also heavily employed in the automotive market, where governmental safety standards demand the most intelligent and advanced safety systems. As shown in Chapter 2, ADAS

applications require very-high computing capabilities, involving the processing of information coming from numerous sensors, such as radars and cameras, distributed throughout the vehicle. This usually requires high-performance many-core CPU or Graphics Processing Unit (GPU) architectures for signal processing, which may or may not be specialized to the target operation due to their limited flexibility and programmability.

On the other hand, by using FPGAs, designers can provide specialized circuits that can be more efficient, in terms of flexibility, performance and power with respect to a general purpose processing unit.

Numerous examples can be found in literature, where FPGAs have been used to implement communications controllers and networks complying to the *FlexRay* automotive standard [185, 186], or to implement high performance radar or image signal processing modules providing driver assistance functionalities [179, 189, 215].

## 3.4 Dynamic Partial Reconfiguration

As systems become more complex and designers are asked to provide even more efficient solutions, FPGA adaptability has become a highly investigated aspect. While native *Flash-based* and *SRAM-based* FPGAs can provide the flexibility of in-field or in-application reprogramming, novel design strategies are required in order to reduce board space, power consumption and resource utilization.

In this context, modern *SRAM-based* devices provide *run-time* or *dynamic partial reconfiguration* (DPR) features. DPR represents the ability to run-time change the functionality implemented by selected portions of a circuit while maintaining the rest of the design in a fully operating state. This extends the native flexibility of FPGAs and the hardware design space allowing to time-multiplex hardware resources. By exploiting DPR, designers can reduce cost and board space since they can fit more logic into the target device, they can include additional functions at run-time, and reduce power consumption by choosing a smaller device or swap out high-performance modules when not needed.

In literature, DPR has been extensively adopted to design adaptive and self-adaptive System-on-Programmable-Chips (SoPC) [24], to increase system fault-tolerance and self-repair capabilities [49, 102, 150], or for on-demand hardware acceleration [44].

Both *Altera* and *Xilinx* provide *SRAM-based* dynamically and partially reconfigurable FPGAs, along with dedicated hardware design flows [5, 230]. In the sequel, only the *Xilinx* partial reconfiguration design flow will be discussed and analyzed, since it is the most advanced and adopted in literature, and it has been used during this PhD work.

Figure 3.27 illustrates the working principle of DPR.

At design-time, DPR requires partitioning the system into static and reconfigurable module. Each reconfigurable module (e.g., *A1-4* in Figure 3.27) must then be binded to a physically user-

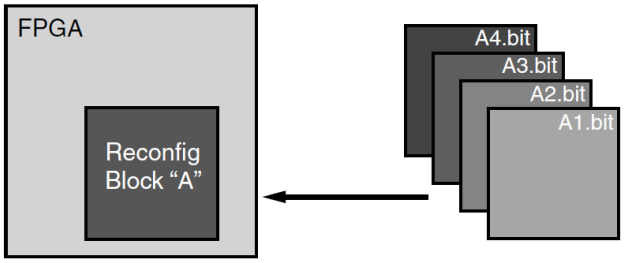



Figure 3.27: Partial Reconfiguration concept [239].

defined reconfigurable region of the FPGA (e.g., *Reconfig Block "A"* in Figure 3.27). Each reconfigurable region can host a single reconfigurable module at a time. Through DPR, reconfigurable modules in a region can then be swapped at run-time without interrupting the operations of the circuit implemented outside the actually reconfigured area.

Xilinx's EDA tools generate separate partial configuration files, called *partial bitstreams* or *partial bitfiles* (i.e., *A1-A4.bit* in Figure 3.27), for each module to be mapped into a specific reconfigurable area.

Figure 3.28 illustrates the partial reconfiguration design flow.

Starting from the *HDL Sources*, the static part of the design and the reconfigurable modules are synthesized separately. The implementation process is then performed in order to produce every possible full design configuration. A full configuration is built merging (i) the *static* netlist, (ii) a reconfigurable module netlist for each user-defined reconfigurable region, and (iii) design constraints files associated to both the static and the reconfigurable modules included in the considered configuration. It is worth to mention that the static portion of the design is implemented only once, and the results are copied in subsequent design configurations (see Figure 3.28).

At the end of the implementation process, the FPGA must be first programmed using a *full bitstream* associated with one of the created design configuration (e.g., *Full_1-N.bit* in Figure 3.28). Afterwards, reconfigurable modules can be swapped at run-time by loading the associated *partial bitfile* (e.g., *RMA.bit*, *RMB.bit* or *RMN.bit* in Figure 3.28) inside the FPGA through one of the available device configuration ports, that allows the access to the device configuration memory.

*Xilinx* devices offer several configuration ports, each one characterized by a maximum bandwidth (see Table 3.1).

DPR approaches can be classified depending on (i) where the *partial bitstreams* are stored and (ii) which configuration port is used to load them. Two main approaches are depicted in Figure 3.29.

Usually, *partial bitstreams* are stored in an external Flash memory. In order to load a partial bit-file, an external or internal controller is needed. The controller must interface to the selected

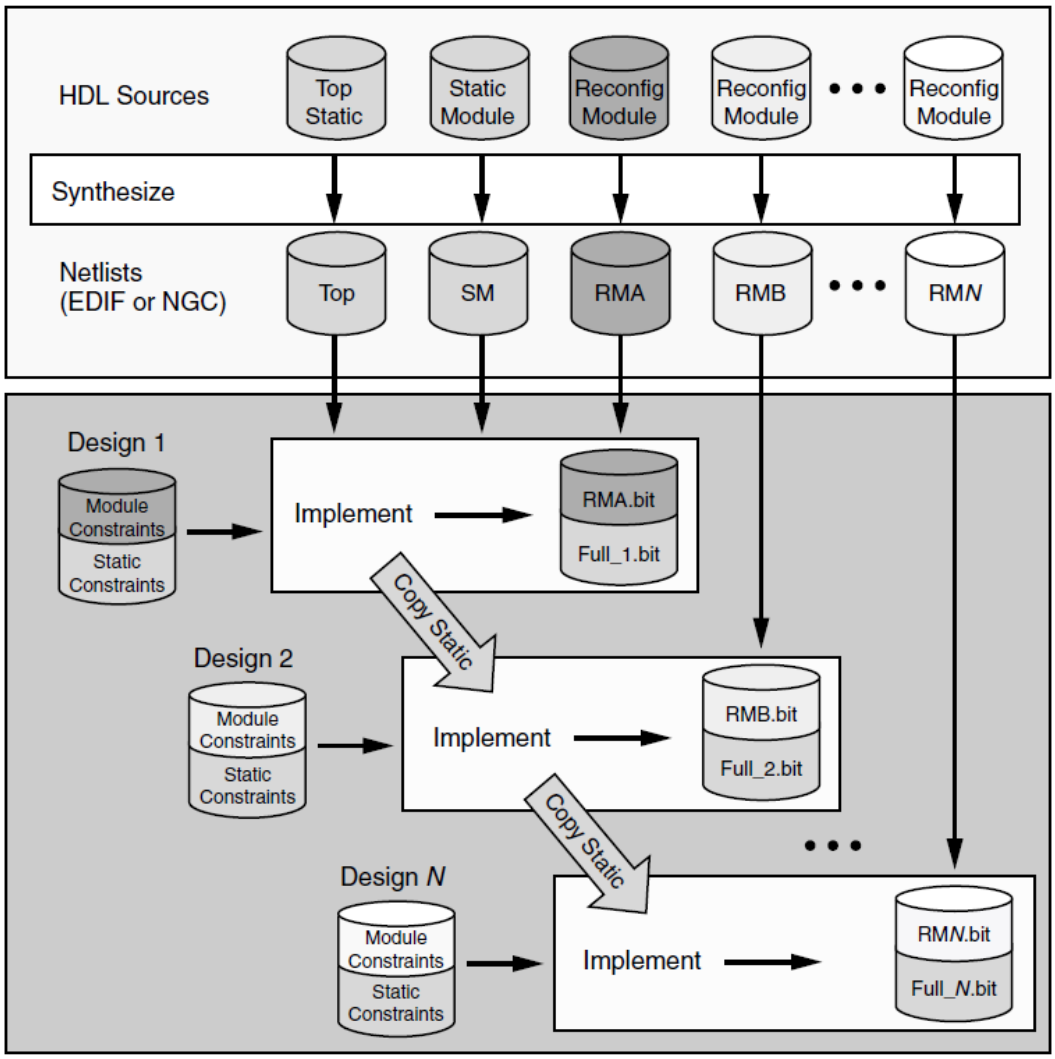



Figure 3.28: Partial Reconfiguration Design Flow [239].

Table 3.1: Characteristics of *Xilinx* FPGAs configuration ports [239].

| Configuration Port | Max. Clock Frequency | Data Width | Max. Bandwidth |
|---|---|---|---|
| *ICAP* | 100 MHz | 32 bit | 3.2 Gbps |
| *SelectMAP* | 100 MHz | 32 bit | 3.2 Gbps |
| *Serial Mode* | 100 MHz | 1 bit | 100 Mbps |
| *JTAG* | 66 MHz | 1 bit | 66 Mbps |

configuration port. In particular, the Internal Configuration Access Port (ICAP) provides access to the FPGA configuration memory from within the FPGA. Consequently, by implementing a config-

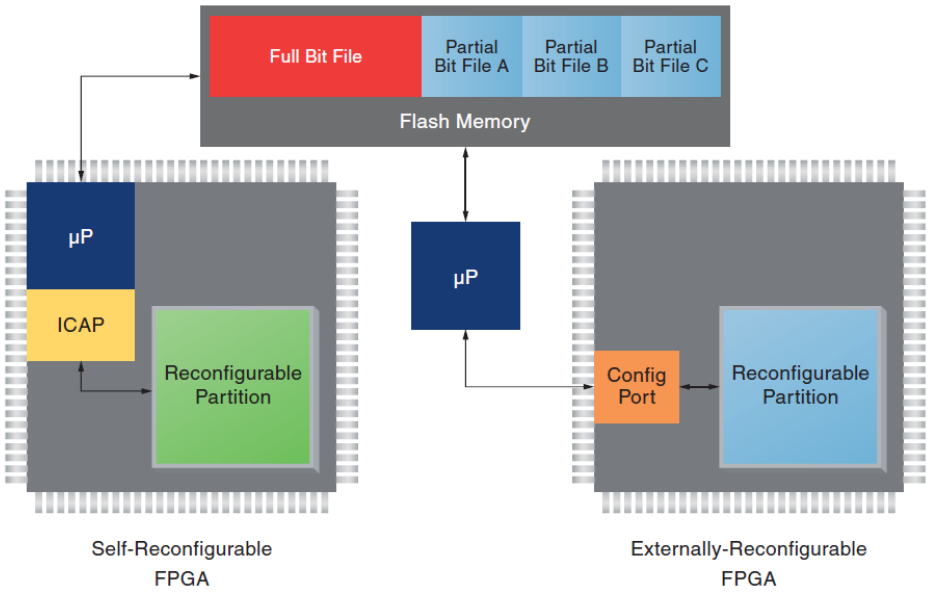



Figure 3.29: Methods for delivering partial bitfiles [230].

uration controller in the FPGA fabric, it is possible to perform the so-called *self-reconfiguration*, i.e., the reconfiguration process is initiated by logic implemented in the actually reconfigured device. As shown in Table 3.1, the ICAP and *SelectMAP* provides the maximum configuration bandwidth, leading to fast reconfiguration times. However, self-reconfiguration solutions are often employed because ICAP eliminates the need of additional external controllers or microprocessors, while guaranteeing the maximum reconfiguration speed [241]. Therefore the ICAP primitive can be instantiated in the HDL description of the design in order to control the transfer of the partial bitstreams before they are sent to the configuration memory.

### 3.4.1 Configuration Details and Bitstream Composition

As aforementioned, a partial bitstream includes all the configuration commands and data necessary to configure a portion of the target FPGA and it can be loaded at any time during normal device operations to replace functionality in a design-time defined FPGA region.

All user-programmable features inside the FPGA are configured exploiting a volatile configuration memory (*SRAM-based*). Values stored in the configuration memory define the LUTs equations, routing networks, embedded memory values and all other aspects of the design.

Xilinx FPGAs' configuration memory is arranged in frames that span the entire device. These frames represent the smallest addressable portions of the device configuration memory space. Nonetheless, loading a partial bitstream into the configuration memory does not require any prior knowledge of the physical location of the reconfigured resources, since the information

needed for frame addressing is included in the partial bitstream itself. Consequently, a valid partial bitstream cannot reconfigure the wrong region of the device.

Figure 3.30 illustrates the process of loading a partial bitstream in the target device. After

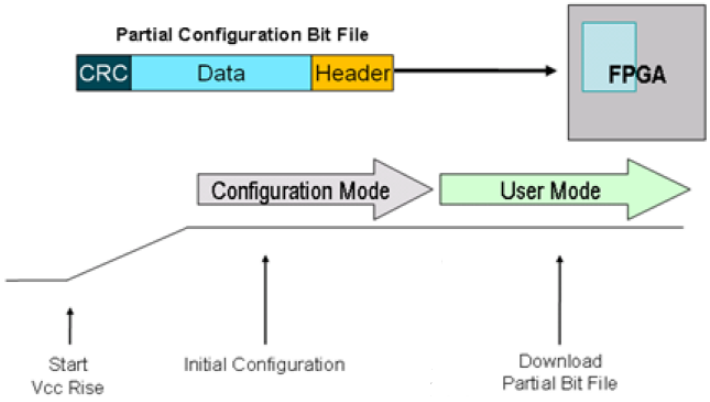



Figure 3.30: Partial Bitstream composition and loading process [241].

power-on, the device configuration memory is empty and put in configuration mode. An initial full configuration is therefore needed. Once the initial configuration is completed, the FPGA enters the user mode, where, at any-time, a partial configuration bit-file can be loaded through one of the available configuration ports.

As shown in Figure 3.30, a typical partial bitstream is composed of the following sections:

- **Header**: this section includes device control and frame addressing information;
- **Configuration Data**: this represent the biggest section of a partial bitstream, consisting of configuration information for all the resources included in the target reconfigurable region. These information may concern LUTs, DSPs, embedded memory blocks, and routing matrices and switches;
- **CRC**: a final 32-bit checksum value is also attached at the end of the bit-file in order to verify the integrity of the actually loaded configuration.

A special partial bitstream is the so-called *blanking bitstream*, which represents an empty reconfigurable module. Basically, by configuring a region with the associated blanking bitstream, the pre-existing functionality will be replaced with an empty module. It is worth to mention that these bitstreams still includes few configuration information for the reconfigurable region I/Os, and for any routing, associated with the static design, that can pass through this region of the FPGA [241].

## 3.5 Dependability issues in modern reconfigurable FPGAs

Dependability aspects must be taken into account when employing FPGAs in safety- or mission-critical applications.

In general, *dependability* can be defined as *"the ability of a system to deliver a service that can justifiably be trusted"*, and *"the ability of a system to avoid failures that are more frequent or more severe, and outage durations that are longer, than is acceptable to the user"* [12, 123]. As stated in [18], *"in case of mission-critical applications we can specify dependability as the capability to tolerate faults induced by the environment that could lead to a failure of the entire system"*.

A *fault* is defined as the misbehavior of an internal component of the system. It may be a physical defect, imperfection, of flaw that occurs within some hardware or software components of the system. Depending on system operations, faults can become *errors* if they are activated and propagated to the outputs of the faulty component. An error is a discrepancy between the intended behavior of a system and its actual behavior inside the system boundary. Eventually, errors can lead to *failures* if the system deviates from the correct service operations or specifications. Figure 3.31 summarizes these concepts illustrating the relationships between faults, errors and failures.


Figure 3.31: Chain of dependability threats.

Faults can be introduced in a system by the user, as example providing wrong inputs, or by the environment. Device aging, temperature, mechanical vibrations, and electromagnetic interference represent some examples of possible faults causes in a digital integrated system [133].

Either operating at ground level or in space, FPGAs and, in general, digital ICs are subject to radiation phenomena, that represents one the most critical environmental aspect and one of the primary causes of faults in modern electronic systems [170, 177, 178]. In fact, the aggressive technology shrinking, voltage scaling and the adoption of Commercial-Off-the-Shelf (COTS) components also in critical applications is exacerbating the issues caused by such phenomena.

Radiation can be defined as the interaction of particles with the electronic device, causing energy exchange. Several kinds of radiations exist, and the main are related to the interaction with *cosmic rays*, *alpha particles*, *electrons*, and *photons*.

The interaction between a ionizing particle and the device causes the so called *funneling* effect.

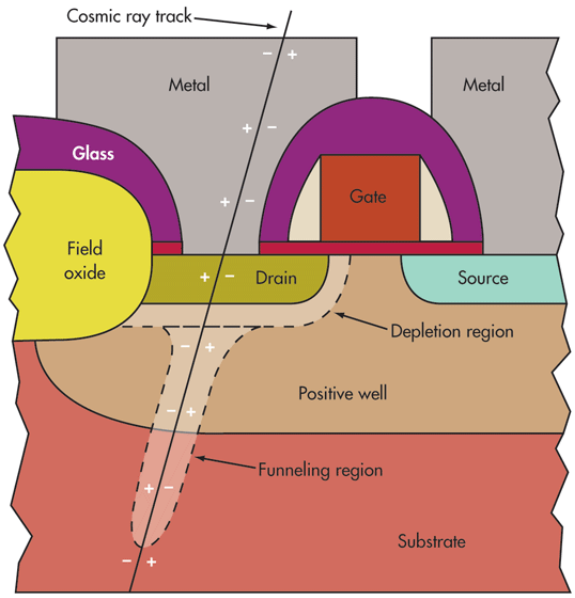



Figure 3.32: Effect of a ionizing particle on a MOS transistor [110].

As shown in Figure 3.32, when a ionizing particle penetrates the semiconductor substrate of a CMOS transistor, it frees electrone-hole pairs, producing a charged track. Consequently, it temporarily modifies the depletion layer of the transistor. The created charges can be collected by reverse-biased junctions and converted into a voltage pulse, thus increasing the probability of a change of information in a sensitive node of the circuit.

A second effect caused by the interaction of ionizing particles with the semiconductor device is the displacement of the crystal lattice. If the particle enters the material, it can damage its crystal lattice by changing the arrangement of the atoms, causing a variation of the electrical properties of the device. Moreover, it is worth noting that the combination of the *funneling* and *displacement* effects can cause an accumulation of charges that can gradually decrease device performances during its operational life.

In order to easily analyze the behavior of the circuits in presence of the aforementioned physical phenomena, *fault models* have been defined. A fault model represents a simplified functional representation of the component's misbehavior, caused by the physical effect.

The two main fault models for radiation-induced faults are the Single Event Effects (SEEs) and Total Ionizing Dose (TID). The former models the effect of the funneling phenomena, while the latter models the effect of crystal lattice displacement and charges accumulation.

As shown in Figure 3.33, SEEs includes all those fault models that can affect the considered device for a limited period of time (i.e., *Soft Errors*), or permanently (i.e., *Hard Errors*).

Soft Errors includes:

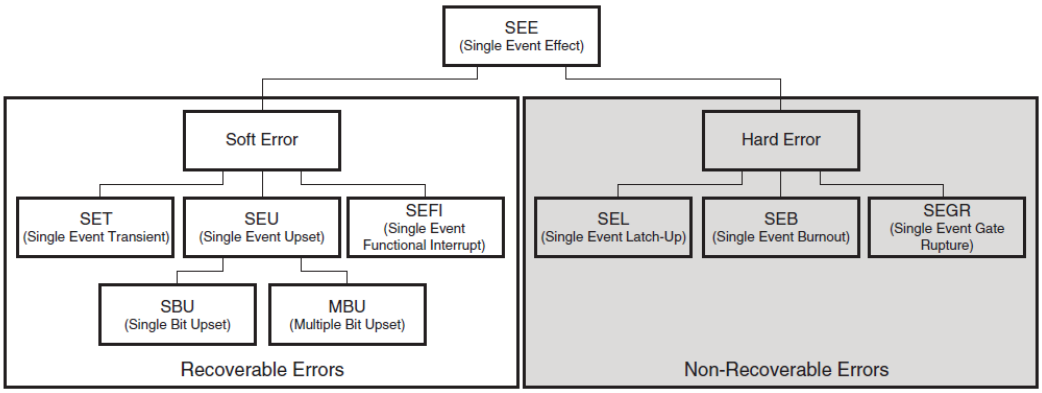



Figure 3.33: Single-Event Effects classification [252].

- **Single Event Transient** (SET): due to the funneling effect, voltage and/or current spikes may be generated for a limited period of time (in the order of picoseconds or nanoseconds). If the pulse-width of this spike is sufficient and at the right time (e.g., close to the sample instant of a Flip-Flop), it can propagate through the logic and eventually sampled by memory elements, thus introducing errors;

- **Single Event Upset** SEU: it represents a change in the state of a memory element (e.g., FFs, SRAM cell, or flash cell) caused by the interaction with a radiation particle. As example, looking at Figure 3.17, if a SET is generated inside an SRAM memory cell internal feedback node, if the duration and width of the glitch is sufficient, it may flip the value stored by that cell. SEUs can be corrected by simply overwriting the content of the affected memory cell.

  In general the upset can affect a single memory element (Single Bit Upset (SBU)) or multiple memory elements (Multiple Bit Upset (MBU)). MBUs can be introduced in the circuit if SEUs are not corrected, thus leading to accumulation effects, or if a single particles has enough energy to upset multiple neighbor memory elements.

  Although in the past the probability of that the system could be affected by an MBU was not relevant, nowadays, the technology shrinking leads to smaller device sizes and higher integration, consequently increasing MBU rates;

- **Single Event Functional Interrupt** (SEFI): it is a temporary errors that affect elements of the device aimed at controlling its functionality. This fault can cause a malfunctioning of the whole device. Usually they can be recovered performing a global reset or a power cycle. As example, faults in the FPGA configuration engine may permanently affect configuration memory and implemented circuit functionality until a device reset or a power cycle is executed.

On the other hand, faults classified as hard errors are:

- **Single Event Latch-Up** (SEL): it is a disruptive permanent error that cause an increase of the device current. An SEL can provoke the destruction of the device itself if not stopped in time by disconnecting the power supply. This phenomenon relies on the presence of parasitic p-n junctions inside the standard CMOS structure. This junctions form a positive feedback network that, if triggered by a current spike, can lead to burning of the device [17];

- **Single Event Burnout** (SEB): Single-event burnout (SEB) is a short-circuiting caused when a high-energy ion impacts a transistor source while the transistor is in its off-state, causing forward biasing. An SEB is typically related to power MOSFETs and high-voltage devices [132];

- **Single Event Gate Rupture** (SEGR): it affects non-volatile memory cells. During writing or erasing operations the transistor is stressed by relatively high voltages. If a heavy particle strikes the transistor gate while it is stressed by such high electric field, it creates a low resistance path within the dielectric between the gate and the substrate. Consequently, this can lead to the dielectric melting.

Beside SEEs, TID are instead caused by spurious charge accumulation and crystal lattice displacements in the device material. This phenomena usually lead to a gradual decrease of device performances, and to an increase of power consumption. These effects are mainly due to the change of electrical characteristics of the device, such as the variation of the transistor threshold that, in case of floating-gate transistors, can lead to a permanent programmability loss.

Considering reconfigurable FPGAs, *SRAM*-based are more susceptible to radiation phenomena with respect to *Flash*-based, mainly due to the smaller transistor technology sizes [17, 145]. It has been also demonstrated that, in modern *SRAM*-based FPGAs, the probability of SETs, SEFIs and hard errors is extremely low if compared to SEUs [126, 226, 252], making the latter a major concern.

Focusing on *SRAM*-based FPGAs, memory elements susceptible to SEUs belong to the following two categories:

1. **user**: this category includes all the elements that can be employed by the user in order to implement the FPGA application (i.e., flip-flops, embedded memories and shift registers beloging to the FPGA fabric or to the embedded hard macros, such as DSPs and high-speed transceivers.

   An SEU in a design memory element cause transient faults that can propagate wrong values through the logic, consequently causing errors and/or failures.

2. **configuration**: this includes all the memory cells and registers that compose the FPGA configuration memory and the associated control circuitry.

If an SEU provokes a bit-flip in the device configuration memory, it could cause a permanent change in the implemented circuit functionality. Such faults are more critical with respect to the ones afecting user memory elements since the FPGA configuration is usually written once only after power-on.

As stated in [187], which takes into account results reported in [4], *"assuming that all flip-flop cells are used, the chance of an upset in these elements is very low if compared to a design that heavily utilizes embedded block RAM (BRAMs) blocks. The probability of an upset is more than 400 times higher than for a flip-flop upset due to the high ratio of BRAM cells to flip-flop cells. For the configuration memory cells, the ratio is even larger..."*. Consequently, SEUs that impact the device configuration memory (and, to some extent, the embedded memory blocks) are the primary sources of soft errors that require mitigation in modern FPGAs [252], and an analysis of their effects on the system is therefore required in order to be able to take the proper countermeasures.

Configuration memory values define the routing architecture and the configuration of all the functional units inside the device (e.g., capture clock and reset polarity for flip-flops, or content of LUTs). Consequently, an SEU in the configuration memory may alter the topology of the implemented architecture. Figure 3.34 shows an example of the effect of a bit-flip in a SRAM memory bit associated to the configuration of a programmable routing matrix.

Figure 3.34 illustrates a fault-free reference configuration of the routing matrix and a possible faulty configuration due to an SEU. In the considered example, the *Net_1*, connecting the input *IN_0* coming from another resource to the output *OUT_1* is replaced with a new net (i.e., *Net_2*, therefore breaking the signal path. Moreover, *Net_2* represents an antenna since *IN_6* is left unconnected and *OUT_1* is driven with an unknown value.

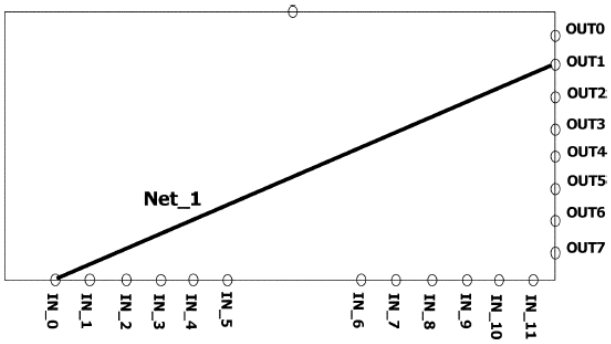
### 3.5.1 Dependability issues in dynamically reconfigurable systems

As discussed in Section 3.4, state-of-the-art *SRAM*-based device offer run-time DPR features. Although DPR can be used to increase reliability figures of a system, e.g., by implementing FPGA configuration memory scrubbing or advanced adaptive fault tolerance and fault recovery mechanisms [49, 102, 150], its adoption in applications demanding high reliability is still limited.

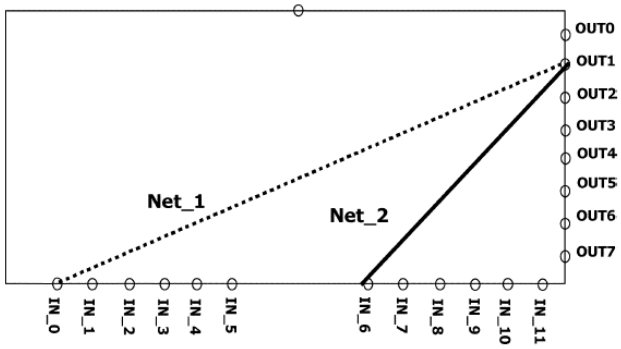
The motivations are related to the dependability of the DPR process itself. In fact, DPR exposes the system to faults and errors affecting both the hardware controller, employed for managing reconfigurations, and actual configuration data (i.e., *partial bitstreams*) that are used to overwrite portions of the FPGA configuration memory content at run-time.

Both types of errors are very critical since, similarly to an SEU effect, a mis-reconfiguration can lead, in the worst case, to a permanent disruption of the entire system functionality. Recovering from such errors could require a full device reconfiguration and/or reset of system operations.

It is worth to mention that writing the FPGA configuration memory with a faulty bitstream can impact not only the functionality implemented by the targeted reconfigurable module, but




(a) Fault-free routing matrix configuration.




(b) Faulty routing matrix configuration.

Figure 3.34: Example of the effect of a SEU on the configuration of a programmable routing matrix [37].

also the static portion of the design due to corrupted frame addressing information or because some interconnections related to the static portion of the design route through the reconfigurable module FPGA region.

As mentioned in Section 3.4, DPR is often carried out through the internal ICAP interface. Reconfiguring through the ICAP does not require any additional external hardware, since reconfiguration tasks can be triggered, executed and monitored by control logic or soft microprocessors embedded in the same FPGA. Essentially, the reconfiguration control logic is in charge of retrieving partial bitstreams from an internal or external memory and delivering them to the ICAP. Clearly, when partial bitstream files are stored locally, the reconfiguration process is faster and more reliable, but, a wide portion of the FPGA embedded memory block is wasted. On the contrary, if partial bitstream files are stored in an external memory, no internal memory is required but the overall reconfiguration process may be slower, with intrinsic lower dependability [191]. As design size and complexity continues to grow (e.g., multi-core or many-core SoCs), area is

becoming a critical resource [183]. Therefore, storing partial bitstreams on external memory devices is the mostly adopted solution. Nevertheless, this approach increases the error probability on partial bitstream files, as communications errors may happen during external data transfer.

As illustrated in Figure 3.30, to validate the integrity of a partial bitstream, each bit file includes a final checksum value. This checksum is compared with an ICAP internally generated one at the end of the reconfiguration process. If a mismatch occurs, the ICAP signals that a corrupted partial bitstream has been received. Nonetheless, this mechanisms does not provide any protection to the system since it is triggered only at the end of the reconfiguration process, i.e., when all data have been already written in the FPGA configuration memory.

Taking into account partial bitstream composition (see Figure 3.30), two kinds of errors may occur: (i) configuration data errors errors and (ii) control and frame addressing errors. When errors occur in the data portion of the bitstream, the recovery procedure is often not critical, because just the reconfigurable region has been corrupted. Loading a new partial bitstream file is in the most cases sufficient to fix the error [231]. However, since routing resources residing in the reconfigurable region can also be used to connect logic resources associated to the static portion of the design, this kind of error can also impact on entire design functionality.

On the other hand, when errors occur in the header section of the bitstream, the corruption modifies control and/or frame addressing information. In such cases, the error may also affect FPGA resources that reside outside the reconfigurable region, therefore a safe recovery requires a full FPGA reconfiguration. Highly reliable or real-time applications must adopt additional error detection mechanisms to overcame this issue.

To conteract the aforementioned issue, an optional and alternative *PerFrameCRC* solution has been introduced by *Xilinx* only for *7-Series* and *Zynq7000* devices [250]. When generating partial bitstreams for these devices, additional intermediate control instructions and checksum values are embedded in the partial bitstream, leading to a bitstream size overhead of about 4-5% [250]. In these devices configuration data are internally buffered by the ICAP, and associated checksums are checked before actually writing them in the configuration memory. Although this method provides a more reliable solution (only available in latest device families) w.r.t. the previous one, these checks are performed with a fixed granularity, leading to error detection capabilities, increased bitstream storage requirements and timing overheads that may not be compliant with the constraints imposed by the target applications.

Nevertheless, in any case, delivering to the ICAP a wrong or a corrupted bitstream can potentially interrupt system operations or permanently change circuit functionality. Either stored in an external memory or in embedded FPGA memory blocks, partial bitstreams are subject to errors. It is therefore essential to monitor configuration data and provide the requested uncorrupted bitstream at ICAP inputs. Moreover, unless a radiation-hardened FPGA is employed, configuration data monitoring alone is not sufficient to overcome the aforementioned issues if it is not implemented in conjunction with a reconfiguration controller that is protected with respect to faults

affecting its hardware resources [71].

Depending on the employed device, bitstream storage and application requirements, different protection schemes can be adopted to increase the overall dependability level of the reconfiguration process. These considerations will be addressed in Chapter 5, that will present some methodologies and tools, developed and analyzed during this PhD work, for enhancing DPR dependability.

CHAPTER 4

# BUILDING ROBUST HARDWARE ACCELERATORS AND SYSTEMS FOR REAL-TIME EMBEDDED IMAGE PROCESSING ON RECONFIGURABLE FPGAS

Considering the applications introduced in Chapter 2, such as automotive pedestrian detection and forward collision avoidance, or space video-based relative navigation, real-time performances of the digital image processing systems are required, in order to quickly react to external dangerous or unwanted events. A *real-time digital image processing system* can be defined as a system *"that regularly captures images, analyses them to obtain some data, and then uses that data to control some activity, that"*, to avoid failures, *"must occur within a predefined and limited time"* [14].

The real-time performance constraints, often merged with the high computational workloads generated by digital image processing algorithms, represents a challenge when trying to build real-time and embedded imaging systems.

Modern embedded processors, providing inherently serial processing capabilities, cannot fulfill the required constraints. Therefore, *hardware acceleration*, i.e., resorting to custom hardware components to significantly speed-up computationally intensive software algorithms, represents a crucial solution.

Looking at common image processing algorithms flows (see Figure 2.1), it is clear that hardware *pipelined* architectures can bring advantages, mainly in terms of throughput, with respect to serial software processing approaches. Depending on the operations performed by each algorithm it may be also possible to parallelize operations and adopt stream processing-based approaches to further increase the performances of the designed hardware system (the reader may refer to [14] for detailed information on the aforementioned techniques).

Due to their flexibility, increasing computing capabilities and resources capacity, FPGAs rep-

resent one of the most popular technology that can be adopted to develop custom hardware accelerators and entire embedded image processing system [14].

Taking into account that the results of the image processing pipeline are used to control the system, when dealing with critical applications it is desiderable, and often mandatory, to guarantee that processing algorithms provide results characterized by a sufficient *"confidence"* level. As discussed in Chapter 2, input images can be affected by unwanted noise or blur, that can hide some information of the target scene. In these cases, pre-processing algorithms are applied to the corrupted input images in order to smooth as much as possible such effects, thus improving and highlighting useful information. This is done to minimize the probability that the subsequent image processing algorithms will extract wrong information.

In addition, environmental conditions changes can cause fluctuations of the contributions given by the unwanted effects, thus altering the performances of the applied algorithms. To overcome this issue and increase image processing algorithms' robustness, the key idea is to introduce self-adaptivity features in order to maintain constant, or improve, the quality of results for a wide range of input conditions, that are not always fully predictable at design-time (e.g., noise level variations). This is also true in space applications, where usually we are not able to fully pre-characterize the environment in which the image processing system is going to be employed. Self-adaptivity has been accomplished by measuring at run-time some characteristics of the input images, and then tuning the algorithm parameters based on such estimations. DPR features of FPGAs have been exploited in order to integrate run-time adaptivity into the designed hardware accelerators.

The following sections present the design of robust image processing hardware accelerators on FPGAs for real-time applications. For increasing algorithm robustness, adaptivity has been taken into account and it has been enabled by mean of dynamic partial reconfiguration of the FPGA design. Referring to the applications and issues discussed in Chapter 4, two case studies will be presented. The former approaches the problem of restoring information from blurred images, while the latter discusses the implementation of a system for extracting and matching features in a robust way.

## 4.1 ABLUR: an FPGA-based adaptive deblurring core for real-time applications

While capturing a frame, the camera must maintain the shutter opened for a finite amount of time, in order to acquire the proper amount of light and take a well defined image. However, as mentioned in Section 2.3, relative movements between the camera and the scene during this interval induce motion blur in the captured image.

It is very difficult to obtain good results by processing blurry frames, and so input images must be firstly enhanced, in order to identify targets or extract information from them.

There exist mechanical techniques to prevent this effect to occur, but they are cumbersome and expensive. Considering for example an Unmanned Aerial Vehicle (UAV) engaged in a save and rescue mission, where recording frames of scene to identify people and animals to rescue is required. In such cases, weight of equipment is of absolute importance, and no extra hardware can be used. In such case, vibrations are unavoidably transmitted to the camera, and recorded frames are affected by blur. It is then necessary to deblur in real-time every frame to allow post-processing algorithms to extract the largest possible amount of information from them.

Restoring the *latent* image from the input blurry one has long been a challenging problem in digital imaging (e.g., [214], [213], [39]). Authors have modelled the task as a two-dimensional *deconvolution* process [33]. This simplification holds on when the blur is considered spatially invariant, meaning that every point in the original image spreads-out the same way in forming the blurry image [190]. In this case, the blurry image is the result of the two-dimensional convolution of the target scene ideal image with the *blur kernel*, also known as *Point Spread Function* PSF (see Section 2.3.1) [35].

However, even in this simplistic case, to accomplish the deblurring task it is necessary to deal with two-dimensional deconvolution, that is well known to be an heavy task [34]. As two-dimensional convolution cannot be directly inverted, it is necessary to perform complex mathematical operations to retrieve the real image hidden behind the blurry input [118]. For this reason, deblurring algorithms are usually unable to achieve real-time performances.

Moreover, very often, to obtain acceptable output results, a tuning phase is required in order to setup the deblurring algorithm parameters. In addition, a new setup phase is in general required when the input images characteristics change (e.g., due to contrast or brightness variations).

The number of research activities dealing with hardware deblurring approaches is very small, above all if it is compared to the huge amount of existing works dealing with software deblurring approaches.

In literature, hardware is usually exploited as a medium to collect more in-depth information about the blurring procedure (e.g., by using sensors to detect the relative camera motion [198] [111]) rather than a way to speed-up mathematical calculations and, consequently, software deblurring approaches.

### 4.1.1 Deblurring Algorithms Overview

One of the very first works on this topic is presented in [136], where an iterative procedure is used for recovering a latent image that has been blurred by a known PSF.

Classical deblurring approaches can be classified in *blind* and *non-blind* algorithms. While the former approach does not need any information about the blur kernel, the latter requires at least an estimation of it. In any case, the problem is severely unconstrained [128].

Early works on deblurring usually model the blur kernel using simple shapes and priors, as in [169]. On the other hand, these exemplifications may lead to poor results when applied to natural images [117]. Linear motion blur kernel model used in many works is very often overly simplified for true motion blurring [32].

During the last years, to consider more complex blurring models, several multi-image based approaches have been proposed. These methods estimate the blur kernel by analysing multiple images of the same scene [38, 174]. Although these approaches have the advantage of discarding too simplistic (and often unrealistic) assumptions, they cannot be applied when it is necessary to work on single input images. For example, [197] presents a hybrid camera system equipped with two imaging sensors. It can simultaneously captures high-resolution video together with a low-resolution one that has denser temporal sampling. Frames captured with higher temporal frequency are less affected by blur, since the smaller camera occlusion time is, the fewer relative movements between camera and scene are. Using the different information retrieved at the same time from the two sensors, it is possible to deblur frames in the high resolution video.

Super-resolution (SR) is an imaging technique that leverages multiple low-resolution frames to construct a high-resolution frame [127]. It involves an exchange of information from frames basing on the assumption that the target has remained unchanged.

The majority of the work published on SR focuses on the mathematical algorithms behind SR and the ability to overcome inherent obstacles such as non-uniform blur [73], and motion estimation errors [76]. However, SR approaches are not suitable when a single standard camera is employed.

An interesting single-image deblurring approach based on *Hyper-Laplacian* priors is presented in [117]. Theoretical basis behind this method rely on the fact that typical gradients distributions in real-world scene images have been proven to be well modeled by a Hyper-Laplacian distribution ($p(x) \propto e^{k|x|^{\alpha}}$), with $0 < \alpha < 1$. However, the usage of such sparse distributions makes the problem more complex, thus slow to solve.

To speed-up the algorithm, authors in [117] present a method that splits the deblurring task into two separated sub-problems. Both these two phases aim at minimizing a cost function to retrieve the most probable latent image. This method proved to be very fast since the most time-consuming computations can be avoided by using a Look-Up-Table-based approach. However, it requires a heavy tuning phase before providing good quality outcomes.

For what concerns deblurring approaches, hardware acceleration has been mainly used for SR [10, 195].

### 4.1.2 *ABLUR* Architecture

This section discusses *ABLUR*, a self-adaptive core, implemented on a single FPGA device, that is able to perform the deblurring task of single input images in real-time. DPR is exploited to

enable self-adaptation of the deblurring algorithm parameters based on input images characteristics. *ABLUR* exploits the algorithm presented in [117], avoiding human interaction during the algorithm parameters tuning phase, by self-adapting to the input images characteristics at run-time.

The approach presented in [117] has been chosen because it has proven to be very fast and accurate; moreover, it is based on the Discrete Fourier Transform (DFT), an operation that is easily implementable in hardware [206]. *ABLUR* architecture is able to deblur single $1024x1024$ pixels images in real-time (i.e., 25 frames-per-second, fps).

As explained in [117], the problem of restoring a latent image $x$, starting from the input blurry one $y$, can be solved in the frequency domain, exploiting Equation 4.1.

$$x = \mathscr{F}^{-1}\left(\frac{\mathscr{F}(-f^1 \oplus w^1 - f^2 \oplus w^2) + \lambda \cdot \mathscr{F}(K)^* \cdot \mathscr{F}(y)}{\|\mathscr{F}(F^1)\| + \|\mathscr{F}(F^2)\| + \lambda \cdot \|\mathscr{F}(K)\|}\right), \tag{4.1}$$

where $\mathscr{F}(Z)$ and $\mathscr{F}^{-1}(Z)$ denote the two-dimensional direct and inverse DFT of a matrix $Z$, respectively [29], and $\|Z\|$ represents the matrix obtained by applying the modulus operator to each element of $Z$.

In Equation 4.1, $*$ is the complex conjugate, $\oplus$ is the convolution operator and $\cdot$ denotes component-wise multiplication (the division is also performed component-wise), while $\lambda$ is a weighting constant. Moreover, since this method belongs to the family of non-blind deblurring algorithms, it requires in input the blur kernel, represented with its Optical Transfer Function (OTF) $K$. The OTF models the transfer function of an optical system, and is represented as a matrix as big as the input image [94].

Instead, $F^1$ and $F^2$ are the OTFs of $f^1$ and $f^2$, that are the two first-order derivative filters in the x and y axis, respectively ($f^1 = [1\ -1]$ and $f^2 = [1\ -1]^T$).

Finally, $w^1$ and $w^2$ are computed as:

$$\begin{aligned} w^1 &= \arg\min_w |w|^\alpha + \frac{1}{2}(w - v^1)^2 \\ w^2 &= \arg\min_w |w|^\alpha + \frac{1}{2}(w - v^2)^2, \end{aligned} \tag{4.2}$$

where

$$\begin{aligned} v^1 &= y \oplus f^1 \\ v^2 &= y \oplus f^2. \end{aligned} \tag{4.3}$$

In Equation 4.2, $\alpha$ is a parameters related to the distribution of the gradients in the input image, and in general it is between $0 < \alpha < 1$ for real-world images, denoting a Hyper-Laplacian ditribution [117], while $\arg\min_z f(z)$ represents the values of $z$ that minimize the function $f(z)$.

Authors propose to solve Equation 4.2 by using a Look-Up Table, which, for a fixed $\alpha$, stores pre-computed data (i.e., $w^1$ and $w^2$), for each possible $v^i$. Obviously, data are discretized, in

order to limit the LUT size. In addition, they propose to compute off-line $\mathscr{F}(K)^*$ and the whole denominator from Equation 4.1 as they do not depend on the input image $y$.
However, this algorithm presents two main limitations:

1. it is a non-blind deblurring algorithm, which implies that the exact blur kernel should be provided as an input parameter to correctly restore an image;

2. it requires a tuning phase that has major impacts on the final produced outcomes, as the value of $\alpha$ has to be fixed, for each input image.

To effectively implement this algorithm on an FPGA device, some considerations and optimizations have been done.

Concerning the first problem, from the knowledge of the system (e.g., vibrations induced to the camera or expected relative motion between camera and the scene), a generic estimation of the blur kernel can be employed as input. Tests have demonstrated that this algorithm is quite robust to errors in the initial kernel estimation, which can be fixed a-priori, and applied on each image at run-time (see Section 4.1.3).

To solve the second problem, it is possible to estimate at run-time the distribution of the input image gradients, characterized by $\alpha$, thus adapting the computations to the actual image scene characteristics.

It is worth noting that, with respect to Equation (4.1), since the OTFs $K$, $F^1$ and $F^2$ are fixed a-priori, the denominator is fully off-line pre-computable thus, at run-time, it can be retrieved from an external memory.

Figure 4.1 shows the overall architecture of *ABLUR*.

*ABLUR* processes a stream of 8-bit packets representing a sequence of 1024x1024 grey scale frames, with 8 bit per pixel (bpp) resolution. It is assumed that the image pixels are received in a raster format, line-by-line from left to right and from top to bottom. *ABLUR* outputs a stream representing the deblurred input frames, with the same bpp resolution.
Several interfaces to external memories are also needed in order to store temporary data, that cannot be efficiently kept in the FPGA internal memory resources.

The following subsections detail all the main modules composing *ABLUR*.

#### 4.1.2.1 Input Image Fast Fourier Transform module (FFT(y))

This module computes the two-dimensional Fast-Fourier Transform of the input image. Since the input image is 1024x1024 pixels, it outputs a matrix of 1024x1024 64-bit complex values (both the *real* and the *complex* parts of each value are represented on 32-bits).

In literature, many real-time FFT hardware modules have been presented (e.g., [51, 207]). Since the focus of this section is not to present an architecture that implements the Fast Fourier

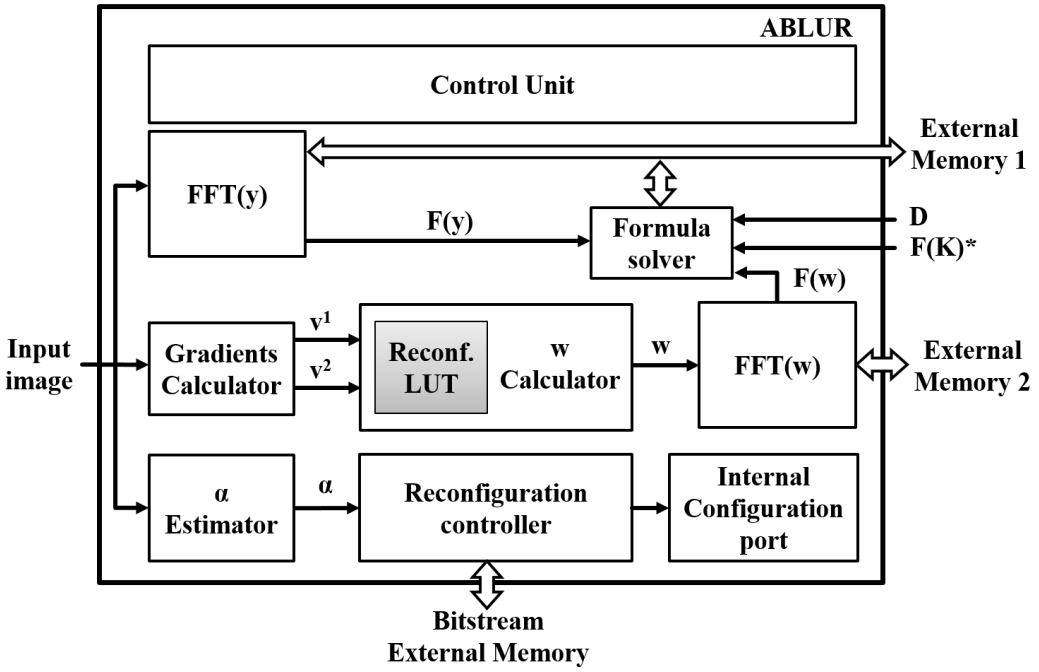



Figure 4.1: *ABLUR* block diagram

Transform, this module has been implemented resorting to the Xilinx *LogiCore Fast Fourier Transform* core [236].

However, to compute a two-dimensional fourier transform, two phases must be performed. First, the FFT is computed for each row of the image, and stored in *External Memory 1*. Then, the final FFT results are computed by retrieving the temporary FFT data in a column order [172]. This module is also in charge of computing the Inverse FFT in Equation 4.1, to extract the deblurred image results.

#### 4.1.2.2 Gradient calculator

This module computes the gradients (i.e., $v^1$ and $v^2$) of the input image by convolving it with the filters $f^1$ and $f^2$ (see Equation 4.3). Figure 4.2 shows the internal architecture of the *Grandients calculator* module.

For each pixel composing the input image, it outputs the associated gradients in the x and y axis (i.e., $v^1$ and $v^2$). Since the input images are received in a row-by-row raster format, and the convolution with the filters $f^1$ and $f^2$ operates on adjacent pixels in the x and y axis, a FIFO buffer is needed to store a single 1024 pixels row of the input image (i.e., *Row Buffer* in Figure 4.2). This buffer has been implemented using a single FPGA internal Block-RAM (BRAM) memory resource [243]. At startup, the FIFO buffer is filled with all the pixels associated to the first row of the

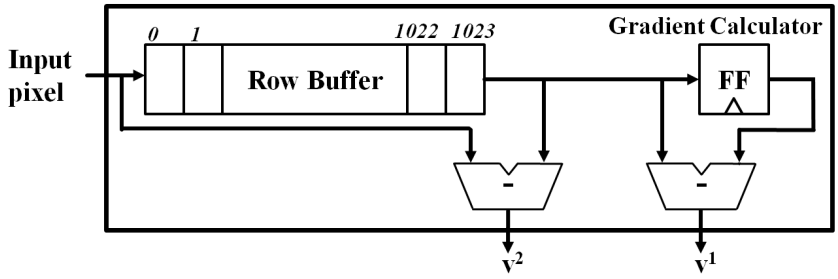



Figure 4.2: Gradient calculator architecture

image. Then, whenever a new input pixels is received, it is stored inside the buffer. Leveraging the dual-port feature of the BRAMs, in the same clock cycle, the oldest stored pixel is read-out. The new read-out pixel is used, in conjuction with the last read-out one, stored in the register *FF* in Figure 4.2, to compute $v^1$. Simultaneously, the read-out pixel is subtracted to the actual received pixel to compute $v^2$.

#### 4.1.2.3 $\alpha$ estimator

The $\alpha$ *estimator* module computes the $\alpha$ parameter (see Equation 4.2) that best fits the characteristics of the input images. The resulting value of $\alpha$ is used to select the right configuration of the *w calculator* LUT, to be applied to the following image. This is acceptable since, at real-time frame rate (i.e., 25 fps), the image characteristics are very similar between the actual frame and the following one.

In particular, $\alpha$ characterizes the gradients distribution of the input image, that, for real-world images, follow a hyper-laplacian distribution (i.e., $p(x) \propto e^{|x|^\alpha}$ where $0 < \alpha < 1$) [117]. The distribution of the gradients can be computed by extracting the gradients histograms.

As shown in Figure 4.3, the $\alpha$ *estimator* is composed by two main sub-modules: (i) the *Histogram Calculator*, and (ii) the $\alpha$ *selector*.

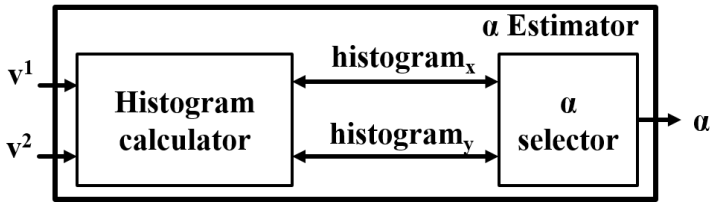



Figure 4.3: $\alpha$ estimator architecture

The *Histogram Calculator* computes the histogram of the input image gradients. Its internal architecture, shown in Figure 4.4, is based on two dual-port BRAM buffers (i.e., $BRAM_x$ and $BRAM_y$), each one associated to a 20-bit counters.

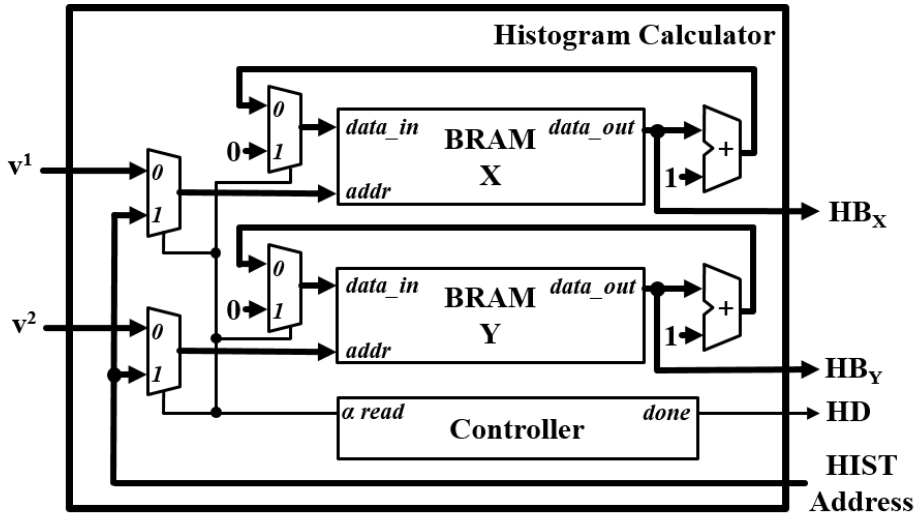



Figure 4.4: Histogram calculator architecture

The values of $v^1$ and $v^2$, received from the *Gradients calculator*, are used to address the two buffers. Within the same clock cycle, the two read-out values are incremented by one, and stored in the same address location of the respective buffer. During this phase the *α read* signal is set to 0 by the *Controller*. When all $v^1$ and $v^2$ values are received, the *Controller* sets the *HD* signal, indicating that the two buffers contain the complete histograms associated to the gradients in the $x$ and $y$ directions.

The *α selector*, using a Look-Up Table (LUT) approach, outputs the $\alpha$ value that best fits the computed histogram distribution. In particular, it contains the $\alpha LUT$, as shown in Figure 4.5, which stores 12 $\alpha$ values in the range (0.40, 0.95), discretized with a step of 0.05. Figure 4.6 plots the hyper-laplacian distributions associated to some $\alpha$ stored in the $\alpha LUT$ and their average slopes in the ranges $[-30,-20]$ and $[20,30]$.

As can be noted from Figure 4.6, looking to the slopes of the functions in the ranges $[-30,-20]$, or $[20,30]$, is sufficient to discriminate between hyper-laplacian functions with different $\alpha$ values. Thus, the *α selector* reads from both histogram buffers the values of the histogram bars, associated to the gradient values 20, -20, 30, and -30, only. To accomplish this task, the *Controller* of the *Histogram Calculator* sets *α read* to 1, while the *HIST Address* signal is used by the *α selector* to extract the histogram bar values $HB_x$ and $HB_y$, associated to the aforementioned gradient values.

Then, the average slope of the hyper-laplacian function in the selected range is computed and used to address the $\alpha LUT$ in order to extract the $\alpha$ parameter (Figure 4.5).

It is worth noting that, although only few values are used, the whole gradients histograms have been computed since these information are often exploited by subsequent image process-

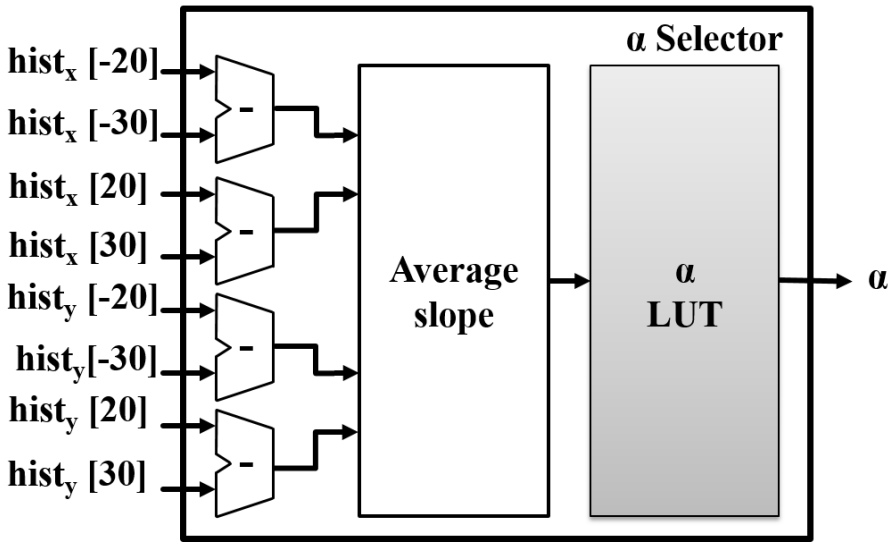



Figure 4.5: $\alpha$ selector internal architecture

ing algorithms (e.g, for edge detection [182]), thus they can be an additional output of *ABLUR*. In any case, this computation does not affect the overall performances, and it requires very few resources.

#### 4.1.2.4 Reconfiguration Manager

The *Reconfiguration Manager* receives in input the $\alpha$, computed by the *$\alpha$ estimator*, and retrieves, from an external memory the partial configuration bitstream associated to the new chosen configuration for the LUT in the *w calculator*. The configuration bitstream is then written to the internal configuration port (i.e., *ICAP* in Xilinx FPGAs), located inside the FPGA device, and directly connected to its configuration engine.

At the end of the reconfiguration process the *w calculator* LUT contains the updated values, corresponding to the selected $\alpha$, that can be used to compute $w$ during the next image cycle.

#### 4.1.2.5 *w* calculator

The *w calculator* module operates in two consecutive steps. First, it solves Equation 4.2 using a LUT approach. Basically, it receives $v^1$ and $v^2$; as the LUT stores the corresponding values of $w$ for discretized $v$ values, it is possible to compute $w^1$ and $w^2$ very fast. For each different value of $\alpha$ a different LUT is required (as Equation 4.2 depends on $\alpha$).

To ensure a good approximation, for a fixed $\alpha$, the LUT contains $10^4$ $w^1$ and $w^2$ 32-bits values, as proposed in [117], leading to the usage of two 312,5 Kbits memories (each one used to compute $w^1$ and $w^2$, respectively).

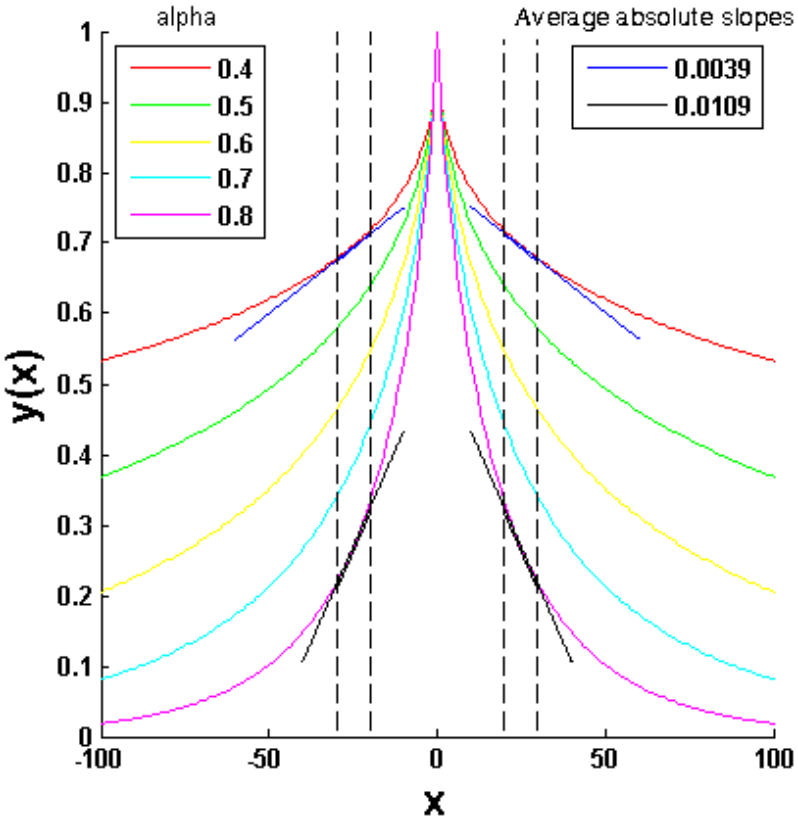



Figure 4.6: Hyper-Laplacian distributions with different $\alpha$ values

In order to save FPGA internal memory resources, at run-time, only the LUT associated with the actual estimated value of $\alpha$ is instantiated inside the FPGA device. Run-time partial reconfiguration is then exploited to change the LUT configuration when the $\alpha$ value changes.

Afterwards, $w^1$ and $w^2$ are convolved with the negated values of the filters $f^1$ and $f^2$, using the same architecture as in Figure 4.2. Finally, the two convolved values are added togheter (see Equation 4.1) to calculate the value of the $w$ that is the output of this module.

#### 4.1.2.6 $w$ Fast Fourier Transform module (FFT($w$))

This module computes the two-dimensional Fast-Fourier Transform of the values received from the *w calculator*. It is important to note that the received values represent a $1024x1024$ elements matrix. As the *FFT(y)* module, the *FFT(w)* has been implemented resorting to the Xilinx *LogiCore Fast Fourier Transform* core [236].

#### 4.1.2.7 Formula Solver

The *Formula Solver* module is in charge of computing the sums and the component-wise division required by Equation 4.1. This module receives $\mathscr{F}(y)$ and $\mathscr{F}(w)$ as inputs, and reads an external memory to retrieve both $\mathscr{F}(K)^*$ and the whole denominator $D$, which are pre-computed off-line (see Section 4.1.2).

This module outputs a 1024x1024 complex matrix on which will be applied the inverse Fourier Trasform to retrieve the deblurred output image.

#### 4.1.2.8 Control Unit

This module coordinates the operations of all the aforementioned modules. In fact, *ABLUR* operations can be grouped in four phases.

During the first phase, while the input image is received, the *FFT(y)*, the *Gradients Calculator*, the *w calculator*, the *FFT(w)*, and the *α estimator* modules are activated. In particular, *FFT(y)* and *FFT(w)* compute the first part of the two-dimensional Fourier Transform, on the rows of the respective input matrices (as mentioned in Section 4.1.2.1 and Section 4.1.2.6), while *α estimator* computes the gradients histograms.

In the second phase, when the image is completely received, *FFT(y)* and *FFT(w)* computes the second part of the Fourier Transforms, retrieving the data computed during the first phase. In the meanwhile, *α estimator* outputs the $\alpha$ value. During this phase, the *Formula Solver* receives in input all the data needed to compute the sums and the division in Equation 4.1.

In the third phase the *FFT(y)* module is used to compute the first part of the inverse Fourier Transform of the values extracted by the formula solver, while the *Reconfiguration Controller* reconfigures the *w calculator LUT* with the chosen configuration, reading the estimated $\alpha$ value.

Finally, in the fourth phase, the same module computes the second part of the inverse Fourier Transform and outputs the deblurred image values.

### 4.1.3 Experimental results

To evaluate the hardware resources usage and the timing performances of the proposed architecture, *ABLUR* has been synthesized and implemented on a Xilinx *Virtex 7 VX485T* FPGA device. Table 4.3 reports the FPGA resources usage of each internal module, along with the percentages of consumed resources with respect to the ones available in the selected device.

From Table 4.3 it is possible to note the limited hardware resources consumption, in terms of both logic (i.e., LUTs and *Digital Signal Processors* (DSPs)) and memory resources (i.e. BRAMs), for the selected device.

The 37 internal memory resources consumed by the *w calculator* are needed to store the *Reconfigurable LUT* associated to the run-time selected $\alpha$ value. The reconfiguration of this Look-Up Table requires $0.2ms$, since the configuration bitstream is about 80 KBytes, and the maxi-

Table 4.1: Resource Usage for *Xilinx Virtex 7 VX485T FPGA device*

| **Module** | **FPGA Area Occupation** | | | |
|---|---|---|---|---|
| | **LUTs** | **FFs** | **BRAMs** | **DSPs** |
| *Control Unit* | 1,347 | 113 | - | - |
| *FFT(y)* | 2,207 | 376 | 16 | 94 |
| *FFT(w)* | 2,207 | 376 | 16 | 94 |
| *Gradients Calculator* | 112 | 35 | 1 | - |
| *α estimator* | 315 | 34 | 2 | - |
| *w calculator* | 265 | 53 | 37 | - |
| *Reconfiguration controller* | 150 | 66 | - | - |
| *Formula Solver* | 2,113 | 560 | - | 4 |
| **Total** | **8,716 (2.87%)** | **1,613 (0.27%)** | **72 (3.50%)** | **192 (6.86%)** |

mum bandwidth of the internal reconfiguration port (called *ICAP* in Xilinx devices) is equal to 3.2 Gbit/s [238]. This reconfiguration time does not influence the overall throughput, since the reconfiguration process can be performed while carrying out the final inverse Fourier transform, that is more time consuming.

At the maximum operating frequency of 255 MHz, *ABLUR* is able to process 29 1024x1024 frames per second, thus achieving real-time performances.

To demonstrate the effectiveness and to quantify the accuracy of the proposed self-adapting approach, a test environment has been developed to read sharp natural-world images and inject motion blur. The proposed architecture has been modeled as a *Matlab* script resorting to a fixed-point algebra to emulate the actual hardware precision. The Matlab model has been used also to perform functional verification of the implemented hardware architecture.

During the test phase, a motion blur kernel was used to simulate relative movements between camera and scene that are 7-pixel long and with an angle of 3 degrees with the x axis. The test environment is based on Matlab, running on Windows 7 x64 on a Notebook PC equipped with an Intel Core i5-2450M @2.50GHz CPU and 8 GB of RAM.

After injecting blur in the original images, the test environment invokes the deblurring function, implementing in software the algorithm executed by *ABLUR*. The blur kernel $k$ passed to the algorithm is a minor perturbation of the true kernel, to mimic kernel estimation errors, as done in [117].

Tests have been performed on 100 1024x1024 pixels images. For each image, a software routine finds the $\alpha$ value that best fits the image gradients distribution. This value is also the one that minimizes the error between the reconstructed latent image, and the original input one. In

particular, to quantify the quality of the reconstructed images, the **RMSE!** (**RMSE!**) has been employed and computed as:

$$\mathrm{RMSE}(L,O) = \sqrt{\frac{\sum_{i=1}^{1024}\sum_{j=1}^{1024}(L_{i,j}-O_{i,j})^2}{1024\cdot 1024}} \tag{4.4}$$

where $L_{i,j}$ and $O_{i,j}$ represent a pixel in position $(i,j)$ in the latent and original images, respectively.

Figure 4.7 shows two images examples along with their hyper-laplacian gradients distributions, characterized by two different $\alpha$ values.

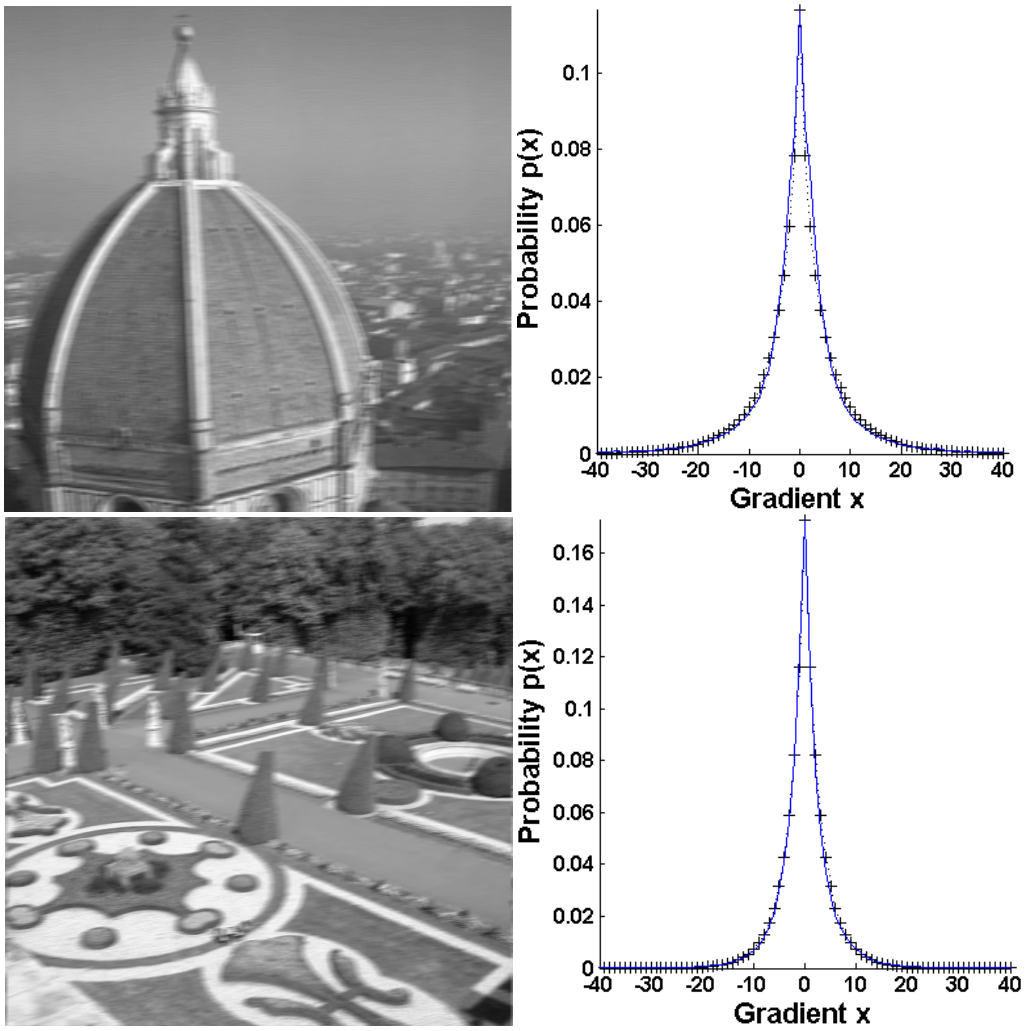



Figure 4.7: Real-world scene images affected by blur and their gradients distribution, together with the Hyper-Laplacian that better fits them (represented with black crosses)

Instead, the graph in Figure 4.8 shows the **RMSE!** results while applying the algorithm implemented in *ABLUR*, with different $\alpha$ values.

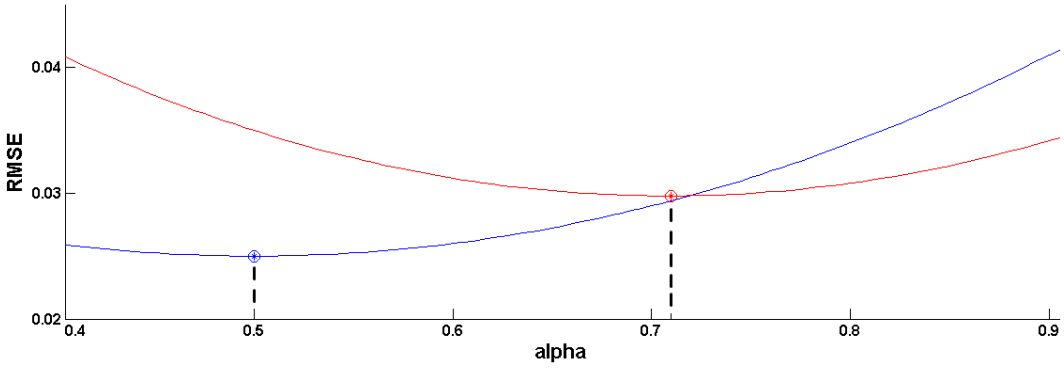



Figure 4.8: RMSE of the recovered latent images w.r.t. the original ones, varying the input $\alpha$ value, for the two examples in Figure 4.7 (the minimum RMSE is highlighted with a circled star)

It can be noted that the optimal $\alpha$ value is different between the two images, and corresponds to the one that characterize their Hyper-Laplacian gradients distribution.

During simulations *ABLUR* was able to identify the optimal $\alpha$ value, with a 0.05 resolution, thus ensuring equals, or even better outcomes w.r.t using a static $\alpha$ input.

In addition, since the hardware implementation of *ABLUR* uses fixed-point data representation, the error introduced with respect to using a software implemented double precision version of the same algorithm has been evaluated. Figure 4.9 shows the visual results and the **RMSE!** values of *ABLUR* and software double precision version outputs.

For the sake of completeness, the output results of *ABLUR* have been compared with the ones obtained by other single-image deblurring approaches (i.e., [117] and two MATLAB built-in functions *Deconvlucy* and *Deconvblind*, both based on the algorithm discussed in [136]). Results are summarized in Table 4.2, and show that ABLUR achieves real-time performances while still providing high quality outcomes. Slight worsening in **RMSE!** are due to approximations of the considered fixed-point algebra. The average elapsed time and the average **RMSE!** are computed over 100 runs.

Table 4.2: Comparison among deblurring approaches in terms of execution time and RMSE

| Algorithm | Avg Elapsed Time (s) | AVG RMSE |
|---|---|---|
| *ABLUR* (HW) | 0,034 | 0,0574 |
| [117] | 2,094 | 0,0409 |
| *Deconvlucy* | 3,126 | 0,0454 |
| *Deconvblind* | 6,396 | 0,0455 |

(a) Latent image restored by *ABLUR*

(b) Latent image restored by double precision software algorithm

Figure 4.9: Example from Figure 4.7 deblurred by *ABLUR* (RMSE=0.044) and by software implemented double precision version of the same algorithm (RMSE=0.039)

*ABLUR* ensures a speed-up of about 60x with respect to the Matlab version of the algorithm proposed in [117], while providing still acceptable results.

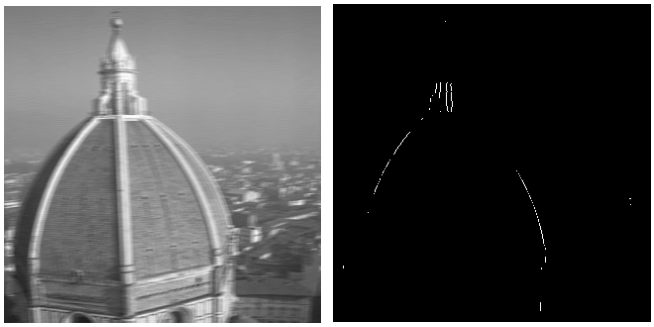
*Deconvlucy* and *Deconvblind* provide similar results in terms of **RMSE!**, while being more time consuming with respect to the approach exploited by *ABLUR* [117].

Finally, a possible example of usage of *ABLUR* is discussed. Consider an UAVs engaged in a save and rescue mission, recording frames of scene to identify people to rescue while flying. To automatically detect people in difficulties, it could be useful to detect edges in every frames; such edges may be compared to typical human shapes, so that an alarm is triggered when possible human target is found. However, in such case, vibrations are unavoidably transmitted to the camera, and recorded frames are affected by blur, so that small edges are confused (or even totally hidden) by blur and impossible to detect. It is then necessary to deblur in real-time every frame to allow post-processing algorithms to extract the largest possible amount of sharp edges from them.

Figure 4.10 shows the outcome of an edge-detection algorithm applied on the original image, its blurry version and the the latent image recovered by *ABLUR*. As is highlighted in this example, edges are definitely more sharp and detailed when extracted from the deblurred image, and very similar to the ones extracted from the original image.

(a) Original sharp image

(b) Edges extracted from original image



(c) Blurry image

(d) Edges extracted from blurry image

(e) Latent image restored by *ABLUR*

(f) Edges extracted from deblurred image

Figure 4.10: Example from Figure 4.7 deblurred by *ABLUR* and edges extracted from blurry and deblurred image

## 4.2 SA-FEMIP: a Self-Adaptive Features Extractor and Matcher IP-core based on Partially Reconfigurable FPGAs for Space Applications

Upcoming plans for solar system exploration in the next 30 years are expected to include landing, sample and return missions to moons, planets, asteroids, and comets [157]. As discussed in Section 2.2, in recent space missions Spirit, Opportunity, and Curiosity, the spacecraft descending trajectory and the final landing point were precomputed and fixed during the mission planning, enabling to reach a maximum landing precision, quantified in terms of area in which the spacecraft is likely to land (landing ellipse) of 20 km [156]. Spacecraft autonomous precision landing capabilities able to reduce the landing ellipse to sub-kilometer accuracy would provide safe and affordable access to landing sites that promise the highest science return and pose minimal risk to the spacecraft [154]. VBN is an area of computer vision that exploits image frames captured by cameras and image processing algorithms to assist navigation in several application domains, including robotics, unmanned vehicles, and avionics [256][155]. The wide availability of cameras on spacecrafts makes VBN a very interesting approach for the implementation of autonomous EDL control systems for next generation space missions.

VBN algorithms extract geometrical information from a set of real-time sampled image frames. As discussed in Section 2.2, they basically perform two activities named *Feature Extraction and Matching* (FEM), and *Motion Estimation*. During Features Extraction and Matching (FEM), each frame is processed to detect those pixels that represent *features* of interest for the image (e.g., corners or edges of surfaces). The detected features are then compared to extract those that can be recognized in two consecutive images (*matching points*). To increase accuracy, motion estimation algorithms require very accurate matching points distributed across the entire frame [134]. FEM algorithms require high computation capability to guarantee high frame rates and therefore high accuracy. Hence, very efficient hardware accelerators for this task are mandatory.

This section discusses *SA-FEMIP*, an optimized FPGA-based self-adaptive FEM architecture based on the well known *Harris* feature extractor algorithm [99]. This architecture introduces self-adaptation of the parameters of the image processing algorithms employed for the FEM task. Self-adaptation enables to better optimize the FEM algorithm to the environmental conditions, thus increasing the robustness with respect to noise and variations of external conditions that are typical of the space environment. Adaptation is obtained introducing very marginal overhead and guaranteeing high operational rates. This is achieved by resorting to DPR capabilities of modern space-qualified FPGAs.

### 4.2.1 Related Works

Feature extraction is the most complex activity performed by FEM algorithms. Several feature extraction algorithms have been proposed in the literature (e.g., Beaudet [20], SUSAN [188], Har-

ris [99], Speeded-Up Robust Features (SURF) [19] and Scale Invariant Feature Transform (SIFT) [135]). From the algorithmic point of view, SURF and SIFT are probably the most robust solutions since they are scale- and rotation-invariant. This means that features can be matched between two consecutive frames even if they have differences in terms of scale and/or rotation. However, due to their complexity, hardware implementations are very resource hungry. As an example, [16] and [27] propose two FPGA-based implementations of the SURF algorithm. The architecture proposed in [16] consumes almost 100% of the LUTs available on a medium sized Xilinx Virtex 6 FPGA, without guaranteeing real-time performances. Similarly, the architecture proposed in [27] consumes about 90% of the internal memory of a Xilinx Virtex 5 FPGA. It saves logic resources, but it is able to real-time process images with a resolution limited to $640x480$ pixels. Another example is presented in [254], where an FPGA-based implementation of the SIFT algorithm is presented. It is able to real-time process $640x480$ pixel images, consuming about 30,000 LUTs and 97 internal DSPs in a Xilinx Virtex 5 FPGA.

Among the available feature extraction algorithms, *Harris* is probably the best trade-off between precision and complexity [204]. Under the assumption of small differences between consecutive frames (i.e., high frame rates or small camera displacements), its accuracy is comparable to SURF and SIFT, with a significant lower complexity. Since high frame-rates are mandatory during the EDL phase to allow real-time correction of the descending trajectory, *Harris* is a very good candidate to implement a high-speed and low-area FEM accelerator block for space-applications [68]. For each pixel $(x, y)$ of a frame, Harris computes the so called *corner response* $R(x, y)$ according to the following equation:

$$R(x,y) = \mathrm{Det}(N(x,y)) - k \cdot \mathrm{Tr}^2(N(x,y)) \tag{4.5}$$

where $k$ is an empirical correction factor equal to 0.04, while $\mathrm{Det}(N(x,y))$ and $\mathrm{Tr}^2(N(x,y))$ represent, respectively, the determinant and the trace of the second-moment matrix, which depends on the spatial image derivatives $L_x$ and $L_y$, in the respective directions (i.e., $x$ and $y$) [99]:

$$N(x,y) = \begin{pmatrix} L_x^2 & L_x L_y \\ L_x L_y & L_y^2 \end{pmatrix} \tag{4.6}$$

where $L_i$ is a spatial image derivative in the direction $i$.

Pixels with high corner response have high probability to represent a corner (i.e., an image feature) of the selected frame and can be selected to search for matching points between consecutive frames.

### 4.2.2 SA-FEMIP Architecture

This section analyzes the SA-FEMIP architecture discussing where and how adaptation to environmental conditions has been introduced.

SA-FEMIP is a pipelined architecture that processes a 32-bit input stream representing a sequence of 1024x1024 grey scale frames with 10 bit per pixel (bpp) resolution (see Figure 4.11). Frame size and resolution are those provided by almost all space-qualified CMOS cameras [75]. Images are received in a raster format, line-by-line from left to right and from top to bottom. The output of SA-FEMIP is the set of matching points identified in the processed frames. The SA-FEMIP pipeline includes three main functional blocks: the *Reconfigurable Gaussian Filter*, the *Adaptive Harris Feature Extractor*, and the *Feature Matcher*. Moreover, SA-FEMIP includes an input/output interface to communicate with an external memory used to temporarily store images filtered by the *Reconfigurable Gaussian Filter* and later required during the feature matching step.

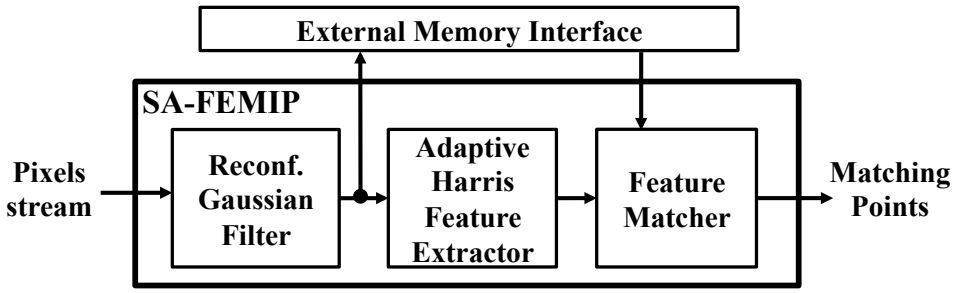



Figure 4.11: SA-FEMIP computational pipeline

The following sections details the operations carried out the modules composing SA-FEMIP, along with their internal architetures.

#### 4.2.2.1 Reconfigurable Gaussian Filter

The *Reconfigurable Gaussian Filter* performs Gaussian smoothing of the input image. It reduces the image noise level, thus improving the feature extraction accuracy [93]. Gaussian filtering is performed by means of a two-dimensional convolution of the input image with a 7x7 Gaussian kernel mask [93] according to Equation 4.7. A 7x7 kernel is enough to approximate a two-dimensional Gaussian function with variance $\sigma_f^2 \leq 2$ [78], which enables to forcefully reduce the noise that strongly affects images taken in space environments.

$$FI(x,y) = \sum_{i=0}^{s-1}\sum_{j=0}^{s-1} I\left(\delta x + i, \delta y + j\right) \cdot K(i,j) \tag{4.7}$$

In Equation 4.7, $FI(x,y)$ is the filtered pixel in position $(x,y)$, $I$ represents the input image, $s$ is the kernel size ($s = 7$ in this architecture), $K(i,j)$ is the kernel factor in position $(i,j)$, and $\delta x$ and $\delta y$ are computed according to the following equation:

$$\delta x, \delta y = x, y - \left(\frac{s-1}{2}\right) \tag{4.8}$$

Figure 4.12 shows the architecture implementing the proposed approach.

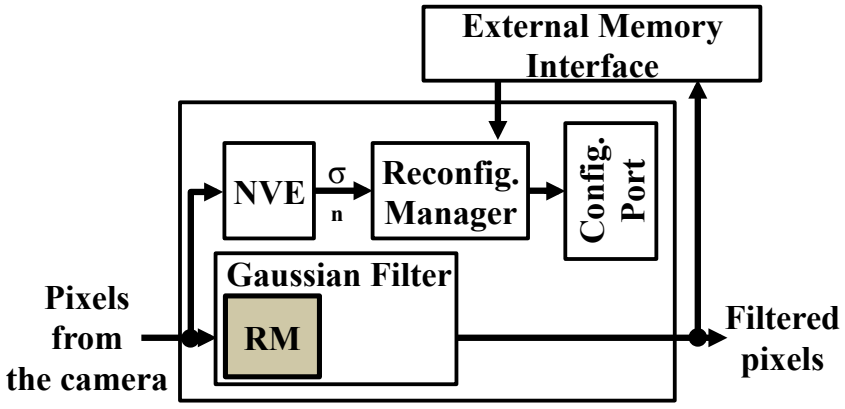



Figure 4.12: Reconfigurable Gaussian Filter hardware architecture

The *Reconfigurable Gaussian Filter* is composed of: (i) the *Noise Variance Estimator* (NVE), (ii) the *Reconfiguration Manager*, and (iii) the *Gaussian Filter*.

Figure 4.13 shows the architecture of the *Gaussian Filter* module.

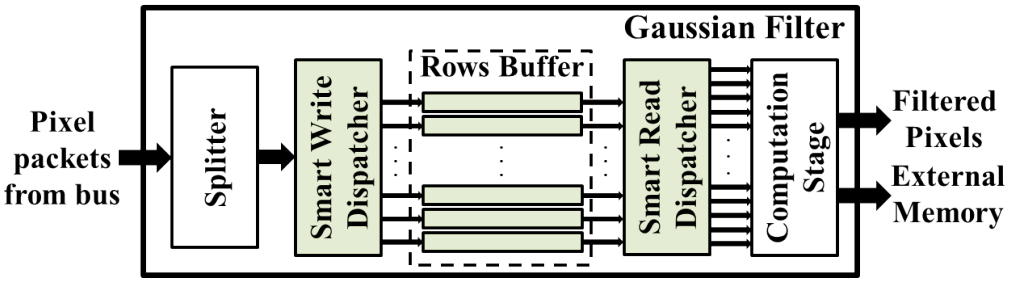



Figure 4.13: *Gaussian Filter* internal architecture

The *Splitter* gets the image flow through the 32-bit input interface and unpacks it in order to reconstruct the original 10-bit pixel flow. Pixels are then sent to the *Smart Write Dispatcher* (SWD), that stores them inside the *Rows Buffer* (RB) before the actual convolution computation.

The *Row Buffer* is composed of 7 FPGA Block-RAMs (BRAMs) [253] each one able to store a full image row[1]. Rows are buffered using a circular policy as reported in Figure 4.14. Pixels of a row are loaded from right to left, and rows are loaded from top to bottom (Figure 4.14(a)). When the buffer is full, the first row of the buffer is used again (Figure 4.14(b)).

When the first 7 rows of the image are ready in the *Row Buffer* the actual pixel filtering starts. At this stage, pixels of the central row (row number 4) can be processed and filtered. It is worth to

[1] The number of rows of the buffer is equal to the kernel size.

remember here that, using a 7x7 kernel matrix, a 3-pixel wide border of the image is not filtered, and related pixels are therefore discarded during filtering.

For each pixel to filter, a 7x7 image patch is extracted from the *Row Buffer* and stored in the *Slide Window Buffer* (i.e., a buffer composed of 49 10-bit registers). This can be efficiently done if one considers that the image is received in a raster way as shown in Figure 4.14(c). At each clock cycle, a full *Row Buffer* column is shifted into the *Sliding Window Buffer* (Figure 4.14(c)). After the 7th clock cycle, the first image block is ready and the *Sliding Window Buffer* is convolved with the *Kernel Mask*. At each following clock cycle, a new *Row Buffer* column enters the *Sliding Window Buffer* and a new filtered pixel of the row is produced. While this process is carried out, new pixels continue to feed the *Row Buffer*, thus implementing a fully pipelined computation. From (4.7), taking into account the considered kernel size (i.e., 7x7 pixels), 49 multiplications are required to produce a filtered pixel $FI(x,y)$. In the proposed architecture, all multiplications are executed in parallel within a single clock cycle. Since kernel factors have been internally represented through constants, 49 constant-multipliers are instantiated. After that, an adder tree (similar to the one presented in [52]) adds the 49 multiplication results to produce the filtered pixel.

The *NVE*, exploiting the algorithm presented in [196], estimates the Gaussian noise variance (i.e., $\sigma_n^2$) affecting the input image. The selected algorithm involves highly parallelizable operations. It first requires to extract the strongest edges (or features) of the input image exploiting the Sobel features extractor. This task is performed using two 2D convolutions [53] between the input image and the Sobel kernels (Equation 4.9) [92]:

$$G_x = I(x,y) * \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}, G_y = I(x,y) * \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$$

$$G = |G_x| + |G_y| \tag{4.9}$$

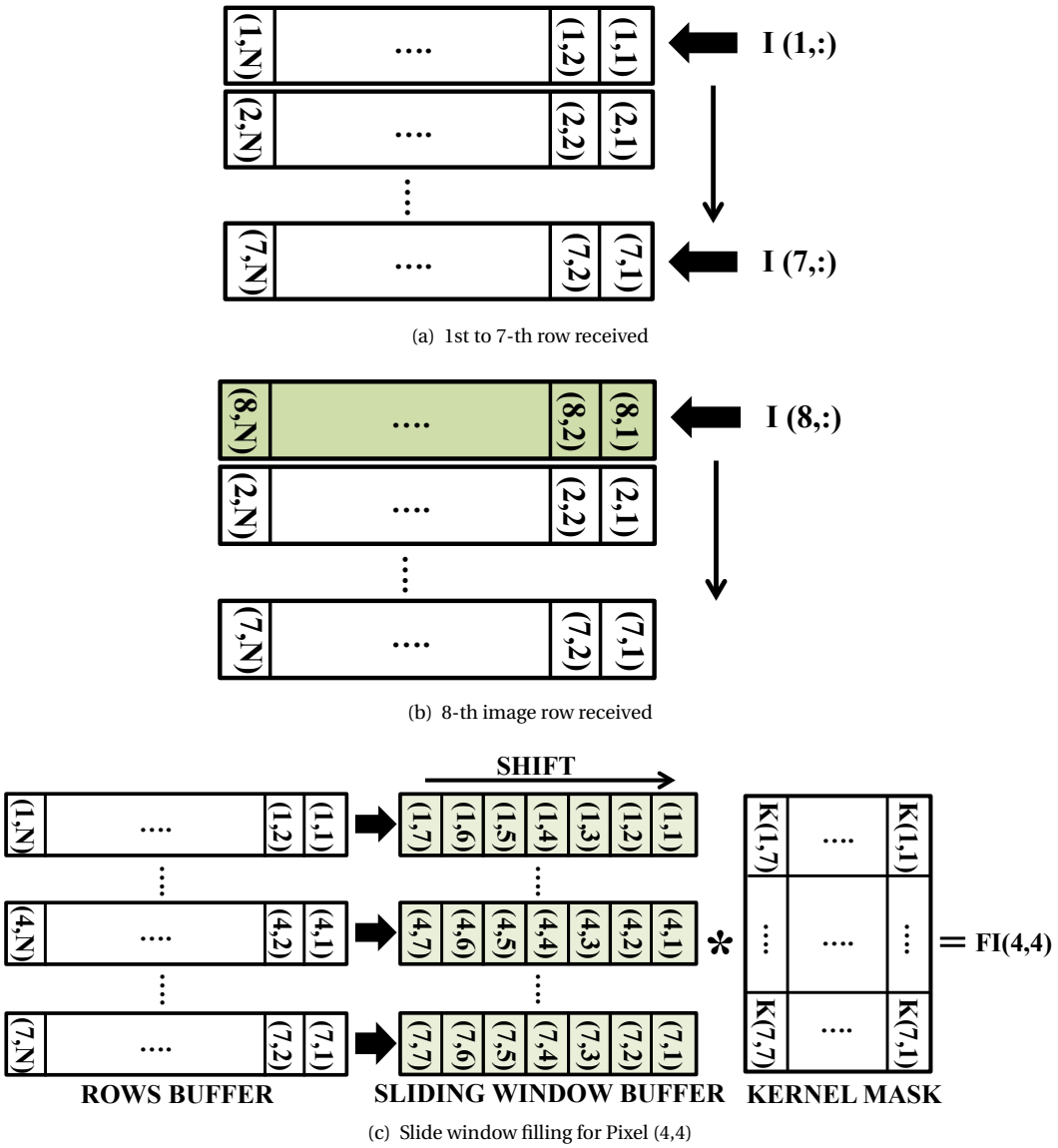
where $I(x,y)$ is the pixel intensity in the (x, y) position of the input image, and $G$ is the edge map associated with the input image. The strongest edges are then extracted by selecting the highest 10% values inside $G$. Finally, $\sigma_n^2$ can be computed as:

$$\sigma_n^2 = \left( C \cdot \sum_{I(x,y) \neq edge} |I(x,y) * N| \right)^2 \tag{4.10}$$

where $N$ is the 3x3 Laplacian kernel [196] and $C$ is a constant defined as:

$$C = \sqrt{\frac{\pi}{2}} \cdot \frac{1}{6(W-2)(H-2)} \tag{4.11}$$

where $W$ and $H$ are the width and height of the input image, respectively (in the proposed architecture $W = H = 1024$).




(a) 1st to 7-th row received

(b) 8-th image row received

(c) Slide window filling for Pixel (4,4)

Figure 4.14: *Gaussian Filter* internal buffers architecture. (i,j) indicates the pixel coordinates.

The internal architecture of the *NVE* is shown in Figure 4.15.

The *SIWB* implements the sliding-window buffering approach of the input image to extract $3x3$ pixel windows*Sobel* using a pipelined architecture similar to the one reported in Figure 4.13.

The outputs of *SIWB* feed the two main modules of *LVE*: the *Sobel Extractor* (SE in Figure 4.15), and the *Laplacian*.

Basically, SE extracts the features from the input image and asserts its output flag only if the

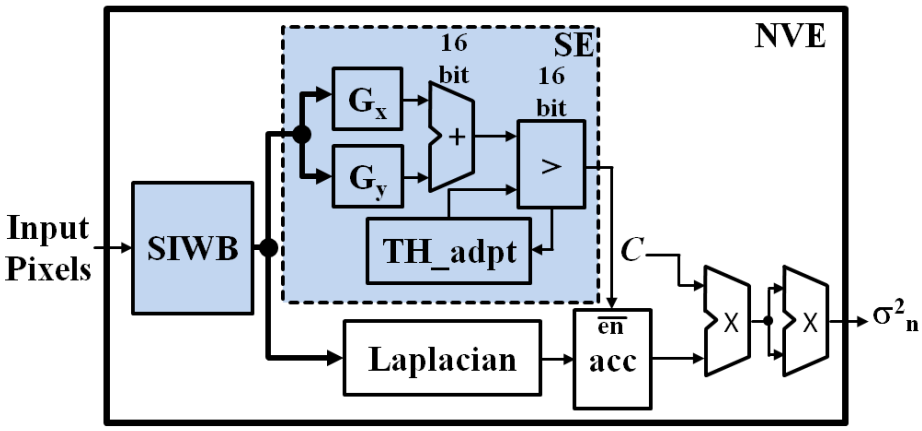



Figure 4.15: *NVE* internal architecture

currently processed pixel is one of the 10% strongest features in the image. First, it executes the operations reported in Equation (4.9). The $G_x$ and $G_y$ modules receive in input the pixels of the current 3x3 patch and compute the two-dimensional convolutions between the input pixels and the Sobel kernels. These two modules are internally implemented as a MUL/ADD tree composed of 6 multipliers (only 6 values are different from zero in Sobel kernels) and 3 adder stages, for a total amount of 5 adders.

The outputs of the $G_x$ and $G_y$ are then added together, through a 16 bit adder, to find the $G$ value (see Equation 4.9). The computed $G$ is compared with a threshold in order to set the SE output only if the current pixel is one of the 10% strongest features in the image.

The threshold value cannot be determined at design time since it strongly depends on the camera and environment conditions. Thus, the *TH_adpt* module (see Figure 4.15) is in charge of calculating the initial threshold value and adapting it frame by frame, by simply applying Algorithm 1. where *N_target_features* represents the strongest features in the input image (i.e., the

**Algorithm 1** Adaptive Thresholding algorithm

$N\_target\_features \leftarrow 0.1 * size(G)$
$Gap \leftarrow N\_Sobel\_features - (N\_target\_features)$
$Offset \leftarrow Gap * (0.5/3000) * Current\_TH$
**if** $Gap < -3000$ || $Gap > 3000$ **then**
    $New\_TH \leftarrow Current\_TH + Offset$
**else**
    $New\_TH \leftarrow Current\_TH$
**end if**

10% of the complete image). *Gap* is the difference between the current number of extracted Sobel features (*N_Sobel_features*) and *N_target_features*. If the value of *Gap* is less than -3000 or more

than 3000, the current value of the threshold (i.e., *Current_TH*) is incremented or decremented (depending on its value) by one *Offset*. The new calculated value for the threshold (i.e., *New_TH*) represents the threshold to be provided in input to the comparator for the next input image.

Since at high frame rates the image conditions between two consecutive frames are approximately the same, the threshold value calculated from the previous frame can be applied to the current processed frame. This task is performed for every input frame, in order to maintain the number of extracted features around *N_target_features*. Obviously, at startup the *Current_TH* is initialized to a low value, and experiments using a MATLAB implementation of the NVE, applied on the *Affine Covariant Regions Datasets* [1], have shown that *TH_adpt* need a maximum of 8 frames to reach a stable threshold value.

In parallel to the SE operations, the *Laplacian* module computes the convolution between the input image and the 3x3 Laplacian Kernel. This operation is performed adopting the same approach used in the $G_x$ and $G_y$ modules. Although, in this case the MUL/ADD tree is composed of 9 multipliers (all Laplacian Kernel factors are different from zero) and 4 adder stages, for a total amount of 8 adders.

The *Laplacian* output is provided in input to an accumulator (*acc* in Figure 4.15). This accumulator is enabled only when SE provides in output a zero, in other words only when the current processed pixel is not one of the 10% strongest features. In this way, when the complete image has been received *acc* contains the value of the sum in Equation 4.10 (i.e., $\sum_{I(x,y)\neq edge} \left| I(x,y) * N \right|$).

The following two multipliers conclude the computation of Equation 4.10.

The computation of $\sigma_n^2$ is exploited by the *Reconfigurable Gaussian Filter* to implement frame-by-frame adaptation (through *DPR*) of the filter variance $\sigma_f^2$, based on the estimated noise affecting the input images. In fact, an architecture that uses a fixed Gaussian filter variance ($\sigma_f^2$) works well if the noise level of the processed frames is known a priori. As an example, a high filter variance is useful for high noise levels. Instead, for low noise levels the images are oversmoothed, thus reducing the accuracy of the feature extraction and matching modules [93].

In literature, many works propose adaptive filters [175][258][50][202]. Among the proposed approaches, those based on evolutionary algorithms are the most promising, in terms of timing performances and hardware resources usage [66]. Nevertheless, they provide very good results if the processed images are similar to the one used during the training phase of the evolutionary algorithm. Instead, if the received image characteristics (e.g., illumination conditions, tonal distribution, etc.) cannot be predicted, as in the harsh space environment, the filtering performances become very poor [153] [152].

In the the proposed approach the noise level estimated for the current frame is used to select the filter variance that would guarantee optimum filtering results. This filter variance is then used to filter the next input image, allowing adaptation of the filter parameters frame-by-frame during the entire descending sequence. The adaptation of the filter variance is achieved by reconfiguring the 49 constant multipliers required to perform the convolution of the image with

the Gaussian kernel. This significantly saves hardware resources with respect to a solution that uses 49 generic multipliers in which the Gaussian kernel constants are selected using multiplexers driven according to the selected filter variance.

In order to enable reconfiguration, the 49 multipliers are enclosed in an FPGA reconfigurable module (RM in Figure 4.12). A reconfigurable module is a portion of an FPGA design that can be reconfigured at run-time, without impacting the behavior of the rest of the design. While the *Gaussian Filter* processes the input image, the *NVE* estimates the noise level. When a full frame has been processed, the NVE provides the current estimated $\sigma_n$ to the *Reconfiguration Manager*. The *Reconfiguration Manager* exploits this value to look-up into a bitstream address table and to select the proper configuration for the multipliers inside the RM. The multipliers reconfiguration is accomplished by reading the multipliers configuration bitstream from the external memory, choosing the configuration associated with the estimated variance. The bistream is then used to program the reconfigurable module of the FPGA resorting to the FPGA internal Configuration Port (i.e., *ICAP* [238] in Xilinx FPGAs).

Finally, since for each value of $\sigma_f^2$ a configuration bitstream must be stored in the external memory, the range of possible $\sigma_f^2$ must be discretized according to the available external memory space (see Section 4.2.2.5 for detailed information about the size of each bitstream).

#### 4.2.2.2 Adaptive Harris Feature Extractor

The *Adaptive Harris Feature Extractor* implements the Harris corner detector. It processes the filtered pixels, received from the *Reconfigurable Gaussian Filter*, and computes the frame features. Each feature is represented by its coordinates $(x, y)$ in the frame, and by the related corner response $R(x, y)$, computed according to Equation 4.5. The computed corner responses must be thresholded in order to identify those features that potentially represent a real corner. However, the value of the threshold strongly depends on the image environment and condition (e.g., brightness, noise, contrast). To provide a certain level of adaptation, [54] introduced a self-adaptive threshold. The threshold $TH$ is initialized at 0 at startup (i.e., all features are accepted). It is then updated based on the number of features extracted from the current image, and applied to the next frame. In particular, for each frame, the number of selected features ($NF$) is compared with the number of expected features ($TF$), set to a predefined value in order to limit internal buffers size. If the two numbers are equal with a tolerance ($\delta$) the threshold is already optimized. If not, the new threshold is computed as $TH = TH + ((TF - NF) * (0.5/TF) * TH)$. The reader may refer to [54] for additional details.

Computing the threshold for the next frame based on information on the current frame is acceptable thanks to the high frame rate of the proposed architecture, that guarantees marginal differences in consecutive frames. However, if the image presents a single small rugged region, the extracted features, and subsequently the extracted matching points, will be concentrated in

that limited region. This leads to poor information extracted from the input frames, and therefore to errors in the *Motion Estimation* phase. This drawback derives from the usage of a single global threshold for an entire frame. The *Adaptive Harris Features Extractor* (AHFE) component analyzed in this section, implements an adaptive cell-based thresholding that relies on frame partitioning to apply different thresholds to different portions of the frame. This ensures that the extracted features uniformly cover the overall frame.

The hardware architecture of the *Adaptive Harris Feature Extractor* is illustrated in Figure 4.16.

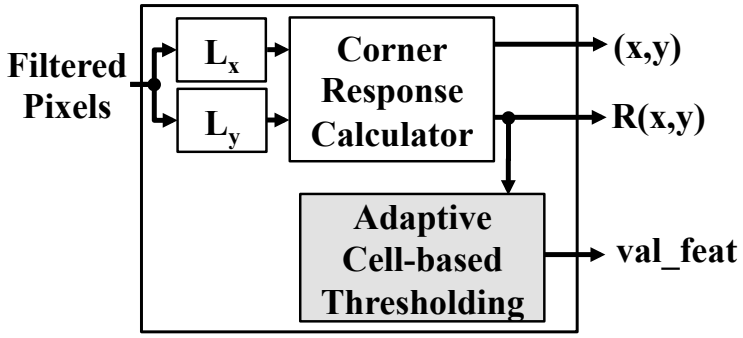



Figure 4.16: *Adaptive Harris Features Extractor* internal architecture

The first two modules of the *Adaptive Harris Feature Extractor*, $L_x$ and $L_y$, compute the spatial image derivatives of the filtered image in the horizontal ($L_x$) and vertical ($L_y$) direction, respectively. This operation is performed by convolving the filtered image, received from the *Reconfigurable Gaussian Filter*, with the 3x3 Prewitt kernel [93], using an architecture similar to the one proposed for the Gaussian Filter. Then, the *Corner Response Calculator* module computes the determinant and the trace of the second-moment matrix $N(x, y)$, which are required to calculate the Harris corner response $R(x, y)$ associated with each input pixel. Finally, the *Adaptive Cell-based Thresholding* (ACTH) module thresholds the computed corner responses, asserting the *val_feat* signal when the current processed pixel is above the threshold and therefore represents an actual feature.

Selecting a well distributed set of features within the frame improves the motion estimation accuracy. In order to level the distribution of the extracted features on the processed frames, the ACTH module splits the input image in 64 cells of 128x128 pixels each. It then tries to extract the same number of features from each cell. This goal is achieved exploiting a local threshold for each cell, instead of using a single global threshold for the overall image. The chosen number of cells represents a good trade-off between accuracy and memory requirements. As it will be discussed in Section 4.2.2.5, it ensures to uniformly cover the input image and, at the same time, to avoid the introduction of a large number of cells that would require a lot of memory to store

**Algorithm 2** Adaptive Cell Thresholding approach

```
Require: NF[8,8]                                   ▷ # extracted features in each cell
Require: TH[8,8]                                   ▷ Threshold value of each cell
Require: TF[8,8]                                   ▷ # target features in each cell
Const N_cell=64                                    ▷ # of cell
Const δ=15                                         ▷ Tolerance
Const LowTH=15                                     ▷ Threshold lower bound
Const OTF=3000                                     ▷ Overall # target features
Curr_EF=∑NF[i,j]                                   ▷ Current overall # extracted features
LowTH_cell[8,8]=[0,...,0]
TF_slack=0
for i=0;i<8;i++ do
    for j=0;j<8;j++ do
        Disp=NF[i,j]-TF[i,j]
        Step = Disp * (0.5/OTF)*TH[i,j]
        if Disp< −δ then
            new_TH[i,j]=TH[i,j]+Step
            if new_TH[i,j]< LowTH then
                new_TH[i,j]=TH[i,j]
                LowTH_cell[i,j]=1
                TF_slack=TF_slack + | Disp |
            end if
        else
            if Disp> +δ then
                new_TH[i,j]=TH[i,j]+Step
            else
                new_TH[i,j]=TH[i,j]
            end if
        end if
    end for
end for
if TF_slack > 0 then
    if TF_slack < N_cell then
        TF_slack_cell = 1
    else
        TF_slack_cell = ⌊ TF_slack ÷ N_cell ⌋
    end if
    for i=0;i<8;i++ do
        for j=0;j<8;j++ do
            if Curr_EF <= OTF then
                if LowTH_cell[i,j]=0 then
                    new_TF[i,j]=TF[i,j]+TF_slack_cell
                else
                    new_TF[i,j]=NF[i,j]
                end if
            else
                if TF[i,j] = 0 then
                    new_TF[i,j]=TF[i,j]
                else
                    new_TF[i,j]=TF[i,j]-1
                end if
            end if
        end for
    end for
end if
return (new_TH, new_TF)
```

the related information items.

The ACTH module analyzes information related to the current frame implementing the decision process described in Algorithm 2, and computes the local thresholds to use for the following frame. The threshold adaptation process requires to know, for each cell composing the frame, (i) the number of extracted features ($NF$), (ii) the current threshold ($TH$) initialized at the highest possible value at startup (i.e., no features are extracted), and (iii) the current number of expected

features ($TF$). In the performed tests, $TF$ has been initialized to 48 to fix the overall number of expected features per frame ($OTF$) to about 3,000 features. This value limits the size of the internal buffer used to store the extracted features in the *Feature Matcher* module. Since $NF$, $TH$ and $TF$ must be defined for each cell of the frame, they are stored in the form of 8x8 matrices, with the matrix elements associated to the defined frame cells.

Algorithm 2 can be split in two main parts. The former (from row 8 to 27) updates the cell threshold values. For every cell $(i, j)$, it compares the number of extracted features $NF[i, j]$ with the number of expected features $TF[i, j]$ ($Disp$ at row 10). If these two values differ no more than a defined tolerance (i.e., the difference is contained in the range $[+\delta,-\delta]$) the threshold is not changed (row 23). Otherwise, the threshold is updated adding $Step$ to its current value (rows 13 and 21). One additional test is performed when the number of extracted features is lower than the number of expected ones (from row 14 to 18). In particular, the updated threshold ($new\_TH[i, j]$) is considered valid if it is higher than a lower bound value ($LowTH$). If not, the threshold is not changed (row 15). This avoids to over-reduce the threshold value and to provide in output weak features that could be potentially associated with the noise in the input frame. In fact, if a cell represents a flat part of the planetary surface, a high value of the image gradient, and consequently a high value of the computed corner response, is mainly due to the noise.

The second part of Alg. 2 (from row 28 to 51) optimizes the number of features extracted for each cell in order to obtain a total number of features for the frame as close as possible $OTF$. To do that, it is worth to remember that all cells that reach the threshold lower bound cannot further update their threshold. If, with this threshold, the number of extracted features for the cell $NF[i, j]$ is lower than the number of expected features for the cell $TF[i, j]$ there is a certain amount of features corresponding to $|\,Disp\,|$ that can be redistributed to other cells with threshold higher than the lower bound. To exploit this, each cell with threshold lower than the lower bound is marked through the $LowTH\_cell[i, j]$ flag (row 16) and the number of unused features of these cells is accumulated in the $TF\_slack$ parameter (row 17) in order to be redistributed to the other cells, according to the decision process described from row 28 to 51. The $TF\_slack$ represents the number of expected features that can be borrowed to the cells that have not reached the threshold lower bound (i.e., $LowTH\_cell[i, j] = 0$). The number of features to borrow to each cell ($TF\_slack\_cell$) is computed dividing $TF\_slack$ by the number of cells composing the image. To ensure a high number of extracted features, the algorithm always borrows at least one feature to each cell with $LowTH\_cell[i, j] = 0$ (rows 29 to 33). If the total number of extracted features $Curr\_EF$ is lower or equal to $OTF$ (from row 36 to 41), if the cell has not reached the threshold lower bound the number of expected features for the cell is increased of $TF\_slack\_cell$ (row 38). Otherwise, it is left unchanged (row 40).

Using this approach, the total number of extracted features ($Curr\_EF$) could increase more and more due to the borrow mechanism, that increases the $TF[i, j]$ values. To allow a decrease of the $TF[i, j]$ values, and so to maintain the overall number of extracted features around $OTF$,

if $Curr\_EF$ exceeds $OTF$, the target feature value of each cell is decreased by 1 (from row 43 to 47).

The hardware architecture of the ACTH module is shown in Figure 4.17.

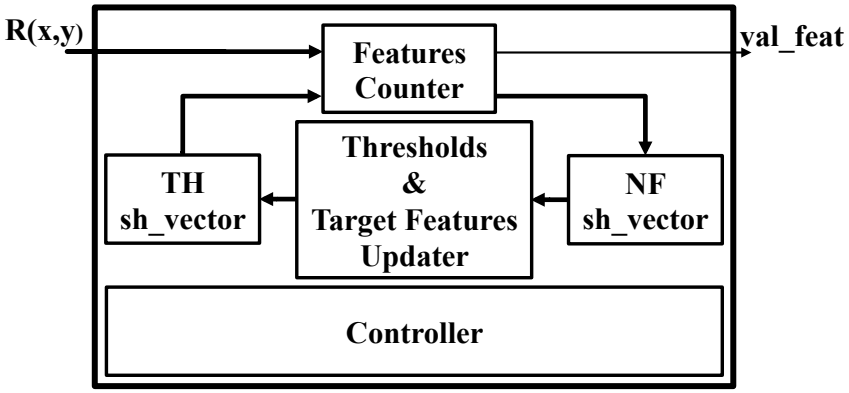



Figure 4.17: Adaptive Cell-based Thresholding hardware architecture

It is composed of four main modules: (i) the *Features Counter*, (ii) the *Thresholds & Target Features Updater*, (iii) the *TH sh_vector*, and (iv) the *NF sh_vector*. The *Thresholds & Target Features Updater* module implements Alg. 2, while the *Features Counter* performs the actual thresholding of each corner response $R(x,y)$ received from the *Corner Response Calculator* (see Figure 4.16). This module reads the thresholds associated with each image cell (i.e., $TH[i,j]$ in Alg. 2) that are stored in the *TH sh_vector*, and compares them with the received corner responses, asserting the *val_feat* signal if $R(x,y)$ is higher than the threshold associated with the image cell containing the currently processed pixel.

The *TH sh_vector* module is implemented as in Figure 4.18. It is composed of eight 8-positions shift registers connected as circular buffers. Each shift register stores eight threshold values associated with a row of image cells (it is worth to remember that the image is split in 64 cells organized in 8 rows with 8 cells each, and a threshold value is associated with each cell). The *en* signal enables the 1-position right shifting operation, while the *Sel* signal selects which shift register must be rotated. These two control signals are driven in order to provide in output the threshold associated with the image cell of the currently processed pixel. Since the image is received in a raster way, and each image cell is composed of 128x128 pixels, *en* is asserted for a clock cycle every 128 received corner responses (i.e., whenever we move from a cell to the following one). Instead, *Sel* selects the next shift register (i.e., the next row of image cells) after 128x1024 received corner responses (i.e., whenever a complete row of image cells has been processed). To avoid loosing the threshold values, during the thresholding phase each shift register composing the *TH sh_vector* acts as a circular buffer through the multiplexer driven by the *th_phase* signal (see Figure 4.17). Instead, during the thresholds updating phase, the content of the *TH sh_vector*

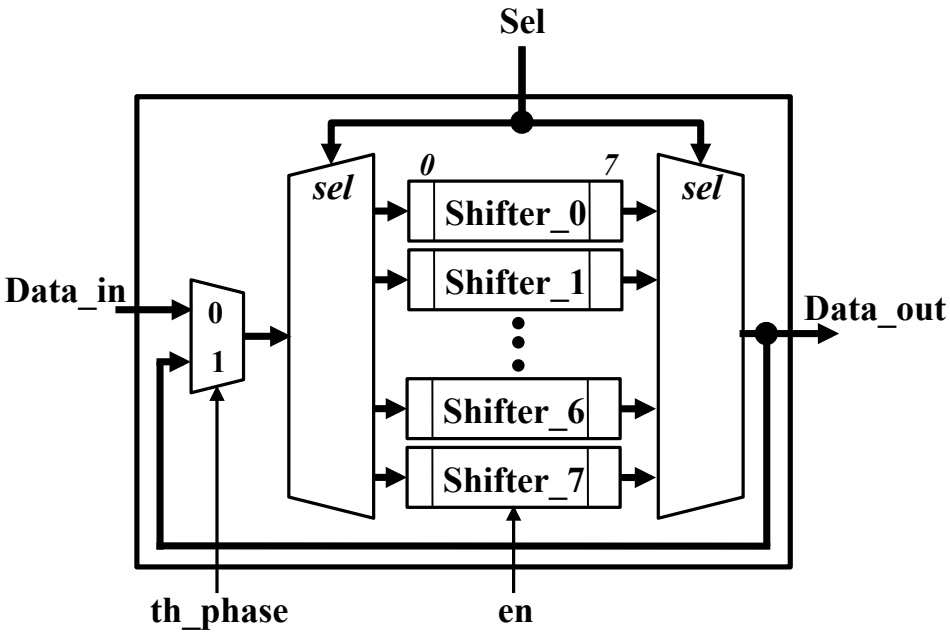



Figure 4.18: TH and NF shifter vector hardware architecture

is overwritten (exploiting the *Data_in* port) with new thresholds values computed by the *Thresholds & Target Features Updater* module.
Simultaneously to the thresholding task, the *Features Counter* counts (through an accumulator) the number of extracted features for each image cell (i.e., $NF[i, j]$ in Alg. 2), and stores these values inside the *NF sh_vector*. The *NF sh_vector* is implemented as the *TH sh_vector* (Figure 4.18), and both modules share the input control signals. Whenever we move from the current image cell to the next one, the content of the internal accumulator is stored inside the *NF sh_vector*, and it is initialized with the output value provided by the *NF sh_vector*. At the end of the operations described by Algorithm 2, a local reset is asserted to clear the content of the *NF sh_vector* in order to prepare it for the next image processing cycle. All aforementioned control signals are generated by the *Controller* module (Figure 4.17), which also coordinates the operations of all modules included in the *ACTH*.

#### 4.2.2.3 Features Matcher

The *Features Matcher* (Figure 4.19) receives the features extracted by the *Adaptive Harris Feature Extractor* and finds the set of features that match in two consecutive images.

This module adopts two different optimization strategies.

The former concerns the matching task, that is performed exploiting un-normalized Cross-

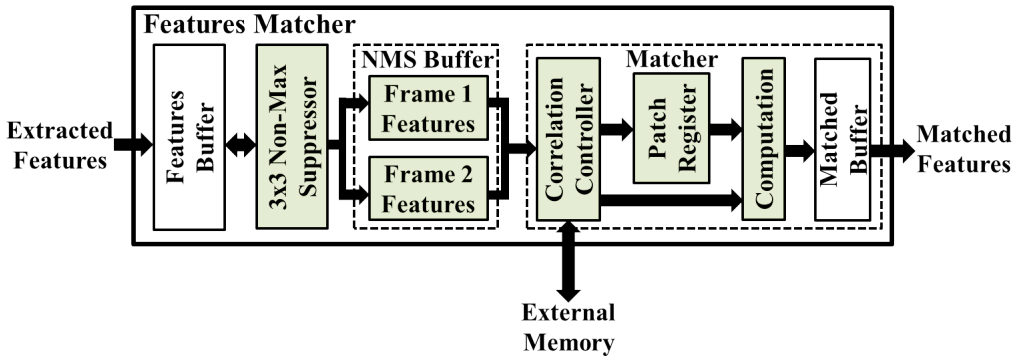



Figure 4.19: *Features Matcher* internal architecture

Correlation:

$$C = \sum_{i,j \in patch} | I_2(i,j) - I_1(i,j) | \tag{4.12}$$

where *patch* identifies the window on which the correlation must be calculated (i.e., correlation window) and $I_x$ identifies the pixel intensity associated with the $x$ image. The less is the value of $C$, the more correlated will be the two points.

The un-normalized Cross Correlation is a matching approach less robust to variations of environmental conditions than the Normalized Cross-Correlation (NCC) one. Although, in this context the high frame rate leads to negligible differences in the conditions (e.g., brightness or contrast) of two consecutive images. Thus, the usage of un-normalized Cross Correlation does not introduce any error in the matching task. In addition, if compared to a NCC approach [257], it leads to a very simple hardware implementation, providing a significant gain in FPGA resources utilization and throughput.

The latter concerns the selection of potentially correlated features. Analyzing the speed of a space-module during the descending phase, and considering the high input frame rate used to sample images, we identified that a feature can perform a maximum movement of 17 pixels between two consecutive images [200]. Thus, two features can be considered as potentially correlated if they are both in a 35x35 pixel neighborhood between the two considered images. Cross-Correlation is therefore computed on these features, only, reducing the computational load and speeding up the matching task.

The *Features Matcher* receives feature coordinates and associated *R-factor* from the *Adaptive Harris Feature Extractor*, and stores them in the *Features Buffer* (FB), implemented as a group of BRAMs.

In this specific implementation, this buffer can store up to 3,500 features, using 14 internal BRAMs.

Whenever an entire image is processed and all features are stored in FB, the *3x3 Non-Max*

*Suppressor* performs a preliminary filtering operation. For each feature, it scans a 3x3 pixels neighborhood looking for close features. If they are found, the feature with the highest *R-factor* in this region is marked as valid, while the others are marked as non-valid. To speed up this operation, that would require a complete search into FB, we observed that, during the whole experimental campaign, no more than 10 features per image row have been identified. Given this observation, considering that features are obtained analyzing the image row by row and then saved into FB, a neighbor feature will be for sure stored in a (+20, -20) region of FB, centered on the considered feature. This allows us to reduce the neighbor search space and therefore to dramatically decrease the execution time, without increasing area occupation.

All valid features are stored in the the *NMS Buffer*, that can store up to 1,000 filtered features coordinates, using 4 BRAMs.

The *NMS Buffer* is composed of two sub-buffers (*Frame 1 Features* buffer and *Frame 2 Features* buffer). These two buffers are alternatively used to internally store features associated with two consecutive images, that must be analyzed and matched. So that no external memory is required to store these information.

The *Correlation Controller* scans the *Frame 1 Features* buffer and the *Frame 2 Features* buffer looking for two correlated features. It compares the coordinates associated with a feature contained in one of the two buffers with all the coordinates in the other buffer. Whenever two potentially correlated features are found (i.e., their distance is no more than 17 pixels, as aforementioned), their un-normalized Cross-Correlation is computed using the intensity of all pixels contained in the two 11x11 pixels windows surrounding the two correlated features. These values (previously stored by the *Gaussian Filter*) are loaded from the external memory.

A 11x11 pixels Cross-Correlation window size has been chosen for the hardware implementation after a test campaign on planetary image sequences, that simulate the descending phase of a spacecraft. For each pair of consecutive frames, the matched features have been saved in order to evaluate the number of true and fake matches. Figure 4.20 shows the maximum rate of fake matches, out of the number of total matches, that was observed between different images, varying the size of the Cross-Correlation window. Results have been evaluated by mean of an automatic script, able to detect fake matches between two consecutive images of the well-known descending test-cases.

As can be seen in Figure 4.20, a window size greater than 11x11 pixels does not provide any significant improvement on the quality of the matched couples. Furthermore, this implementation provides more than 9x precision improvement, compared to the current state-of-the-art [68], which is based on a 7x7 pixels window.

The 11x11 window related to the first feature is loaded into the *Patch Register*, that is composed of 121 25-bit registers. Then, while the window associated to the feature of the second image is loaded, the cross-correlation is computed "on-the-fly". Each time a new pixel is received from the external memory, it is subtracted from the corresponding pixel of the first image, that is

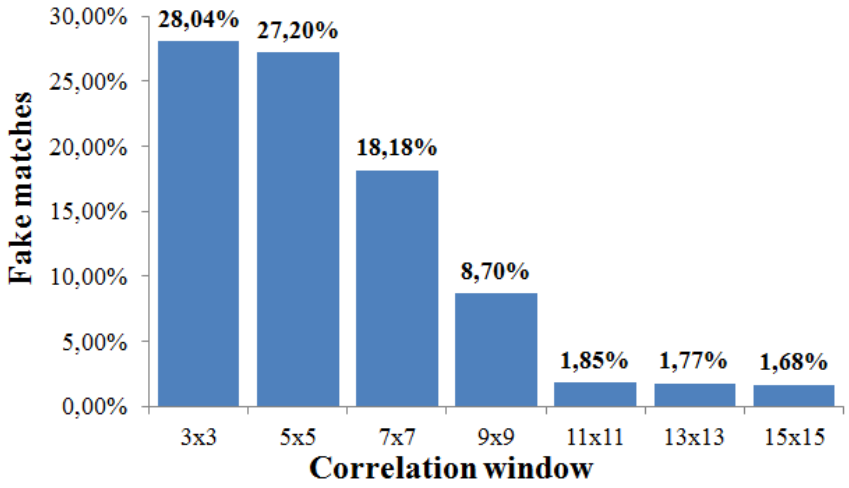



Figure 4.20: Fake matches on test images ranging different Cross-Correlation window size

already stored in the *Patch Register*. This operation is performed by the *Computation* module that contains a 25-bit subtractor connected to an accumulator. This approach makes the area occupation of this module independent from the correlation window dimension, making the designer free to select the more appropriate correlation window without any area occupation penalty.

Finally, the Cross-Correlation results are thresholded, in order to eliminate fake-matchings. If the calculated Cross-Correlation value is less than a given threshold, the coordinates of the correlated features are stored inside the internal *Matched Buffer*, implemented as a single BRAM. This buffer is able to store up to 512 matched features pairs. Moreover, since a feature of the first image can be correlated to several features of the second image, only the match that has the lowest Cross-Correlation value (i.e., the highest probability to be correlated) is considered valid. This ensures unique matched pairs, and higher quality of matches.

#### 4.2.2.4 SA-FEMIP timing diagram

During the reconfiguration process, the *Reconfiguration Manager* in the *Reconfigurable Gaussian Filter* must access the external memory to retrieve the RM configuration bitstream. To avoid the stall of the processing chain, this access must be scheduled when no other module requires information from the external memory. As shown in the timing diagram of Figure 4.21, the external memory is accessed by the *Reconfigurable Gaussian Filter* in write mode to store the computed filtered pixel values. During this phase the *Reconfigurable Gaussian Filter* and the *Adaptive Harris Feature Extractor* work in pipeline, while the noise variance is computed (Image Filtering, Features Extraction and Noise Estimation activities in Figure 4.21). At the end of the feature extraction, the *NMS* phase takes place, and, finally, the *Feature Matcher* performs the *matching* phase where it accesses the external memory in read mode to retrieve the data needed to com-

pute the cross-correlation. It is worth noting that the Image Filtering and the Features Extraction slots are not perfectly aligned due to the latency in loading the internal pipeline of the *Reconfigurable Gaussian Filter*. Looking at Figure 4.21, the external memory is always idle during the NMS phase ($t_{idle}$ in Figure 4.21). This time slot can be used to reconfigure the filter (R task in Figure 4.21) without stalling the processing chain. This means that no timing overhead is introduced in the feature extraction and matching task.

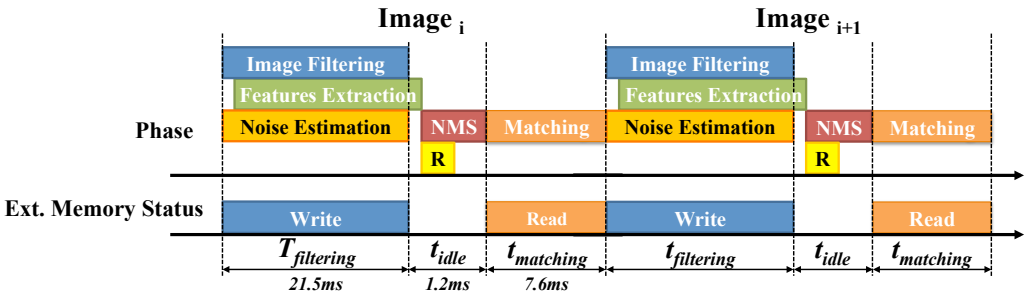



Figure 4.21: Timing diagram of SA-FEMIP

#### 4.2.2.5 Experimental Results

To estimate the hardware resources and the timing performances, the SA-FEMIP architecture has been synthesized on a space-qualified Xilinx *Virtex 4-QV VLX200* FPGA device that, together with the *Virtex 5-QV VFX130* FPGA, represents the state-of-the-art architecture for space-qualified reprogrammable FPGAs. The reason to select the Virtex 4 architecture instead of the newer Virtex 5 is twofold. First the SA-FEMIP architecture has been designed to be integrated and tested inside the *Thales Alenia Space Avionic Testbench* (ATB), i.e., a hardware infrastructure emulating the on-board computing platform of a spacecraft. The ATB is equipped with a *Gaisler Research GR-CPCI-XC4V* development board [86]. This board integrates a Xilinx *Virtex 4 VLX200* FPGA, which provides the same internal logic architecture of the space-qualified version. Second, implementing SA-FEMIP on a Virtex 4 FPGA allowed us to perform fair comparisons with other published architectures, thus highlighting the benefits of the introduced improvements.

Table 4.3 compares the proposed adaptive architecture with the non-adaptive architecture proposed in [54] (FEMIP). Comparison is performed in terms of area overhead by considering internal logic and memory resources (i.e., registers, Look-Up Tables (LUTs), and Block-RAMs (BRAMs) [253]). Percentages in Table 4.3 represent the used portion of the hardware resources available in the Xilinx *Virtex 4-QV VLX200* FPGA. It is important to point out that the synthesis of both *FEMIP* and *SA-FEMIP* architectures has been forced to avoid the use of DSPs. The reasons for this choice will be better elaborated later in this section. Power consumption is analyzed considering an operating frequency of 60 MHz for both architectures.

Table 4.3: Resources usage and power consumption of *FEMIP* and SA-FEMIP, implemented on a *Xilinx XQR4VLX200 Virtex 4 FPGA device*

| | Module | FPGA Area Occupation | | | Max.Freq. | Power |
|---|---|---|---|---|---|---|
| | | Registers | LUTs | BRAMs | [MHz] | [W] |
| **FEMIP**[54] | *GF* | 696 (0.30%) | 5,896 (3.31%) | 7 (2.08%) | 118.36 | 0.064 |
| | *HFE* | 1,106 (0.62%) | 11,081 (6.22%) | 6 (1.79%) | 62.55 | 0.407 |
| | *FM* | 2,432 (1.36%) | 656 (0.37%) | 19 (5.65%) | 101.30 | 0.037 |
| | **Total** | **4,234 (2.38%)** | **17,633 (9.89%)** | **32 (9.52%)** | 62.55 | **2.002** |
| **Proposed** | *RF* | 939 (0.53%) | 7,448 (4.18%) | 10 (2.98%) | 118.36 | 0.083 |
| | *AHFE* | 1,362 (0.76%) | 12,468 (7.00%) | 6 (1.79%) | 62.55 | 0.462 |
| | *FM* | 2,432 (1.36%) | 656 (0.37%) | 19 (5.65%) | 101.30 | 0.037 |
| | **Total** | **4,733 (2.66%)** | **20,576 (11.55%)** | **35 (10.42%)** | 62.55 | **2.097** |
| **Overhead** | **Total** | **499 (0.28%)** | **2,943 (1.66%)** | **3 (0.90%)** | 0 | **0.095** |

Table 4.3 shows that SA-FEMIP FPGA occupation is around 10% for logic and memory resources, and the overhead w.r.t. FEMIP is less than 2%. This overhead is due to the additional modules required to perform adaptation in the *Reconfigurable Gaussian Filter* (RF) and the additional hardware required to implement the *Adaptive Harris Features Extractor* (AHFE). In particular, in the RF, the increased occupation is due to the NVE and the Reconfiguration Manager modules. Instead, in the AHFE, the area overhead is introduced by the usage of a more complex thresholding approach, with respect to the simple one adopted in FEMIP. It is worth to highlight here that an effort has been placed to limit the registers overhead. The AHFE architecture strongly relies on shift registers structures to implement the required vectors and matrices included in Alg. 2. This kind of component can be efficiently implemented in Xilinx FPGAs, exploiting the Xilinx *SRL* capability of the Look-Up Tables (LUTs) [253].

The maximum operating frequencies of each module reported in Table 4.3 demonstrate that no timing penalty is introduced in SA-FEMIP by the introduction of the adaptivity features.

The power consumption of each module reported in Table 4.3 does not take into account the contribution of the clock circuitry and the leakage. These contributions are included in the overall power consumption. By comparing the power consumption of SA-FEMIP with the one of FEMIP a very limited overhead equal to 4.75% is observed. It is worth noting that the power consumption of the RF module does not include the power used during the partial reconfiguration process. However, according to [25] the reconfiguration process consumes few tens of mW, only.

Eventually, the throughput, in terms of frames-per-second (fps), is the same (i.e., 33 fps) for both FEMIP and SA-FEMIP.

In Table 4.4, the performances and the area occupation of SA-FEMIP have been compared with FEIC [68] [69]. FEIC is a Feature Extraction and matching Integrated Circuit, based on the Harris algorithm, that University of Dundee developed for the European Space Agency (ESA) in the framework of the Navigation for Planetary Approach and Landing (NPAL) project. LUTs and BRAMs used by FEIC are reported for a Virtex II device (as in [69]), but the internal logic and

memory architecture is the same as in Virtex 4 family devices. The reported data confirm the great improvements of the proposed architecture, both in terms of resources usage and throughput.

Table 4.4: Resource usage and throughput of *FEIC* and SA-FEMIP for a *Xilinx XQR4VLX200 Virtex 4 FPGA device*

| | **Resource Usage** | | **Max. Speed** |
|---|---|---|---|
| | **LUTs** | **BRAMs [KB]** | **[fps]** |
| **Proposed** | 20,576 | 78.75 | 33 |
| **FEIC** [69] | 50,688 | 162.5 | 20 |
| **Improvements** | **-59.4%** | **-51.5%** | **+65%** |

The low area occupation of SA-FEMIP allow designers to exploit the free hardware resources to apply fault mitigation strategies to increase the reliability of the design, a key requirement in space applications. Several fault-mitigation strategy against Single Event Upset (SEU) can be applied on FPGA devices. Following [205], these techniques can be classified as (1) *netlist level techniques* or (2) *register transfer level techniques*.

Netlist level techniques include different types of Triple Modular Redundancy (TMR) techniques [205]. Triplication can be limited to the sequential elements of the circuit (i.e., *Sequential logic TMR*) introducing for each register of the design two additional registers and a 3-input voter. Otherwise, the full design can be triplicated (i.e., *Global TMR*) introducing a hardware overhead equal to the 200% of the original design.

Register transfer level techniques aim at protecting the Finite State Machines (FSMs) of the design (e.g., *Safe FSM Coding*, and *3-Hamming distance enhancement in FSMs*). Usually, the overhead introduced by these techniques is one order of magnitude lower than the one associated with the TMR techniques.

In general, the total hardware overhead, even if a combination of the aforementioned techniques is exploited, can vary from 60% up to 200% of the original design [205]. It is clear that, given the low amount of resources required by SA-FEMIP, fault tolerance techniques can be freely implemented within the selected device. Moreover, even after the implementation of fault tolerance techniques, space is also available to integrate in the same device additional FPGA-based IP-cores useful to accelerate other computational intensive tasks performed during the descending phase (e.g., *Hazard map computation* [210]). This is very important considering the limited resources available in space applications.

As mentioned at the beginning of this section SA-FEMIP has been synthesized avoiding the use of DSPs. This decision can now be better motivated. DSPs have the advantage of further reducing the area occupation of FEMIP especially when multipliers are implemented. With the use of DSPs the SA-FEMIP occupation would be reduced to 9,029 (5.06%) LUTs, 66 (68.75%) DSPs, while the occupation of registers and BRAMs remains the same. Nevertheless, DSPs are limited

resources. With 66 DSPs required out of the 96 available in the Virtex 4 VLX200 FPGA, TMR techniques for this portion of the design would not be possible. Moreover, the intensive use of DSPs increase the routing complexity.

SA-FEMIP has not been compared to [31], since [31] implements the multi-scale Harris detector (i.e., a rotation-invariant version of the Harris detector). [31] consumes a lot of hardware resources, and implements a feature that is not actually required in EDL applications since rotations between two consecutive images are limited [70].

The proposed architecture has been evaluated in terms of accuracy and robustness, exploiting an image dataset, provided by *Thales Alenia Space Italia s.p.a.* company, that covers different landing zones (i.e., portions of the Mars surface), environmental conditions (i.e., image quality), and camera movement types, in a synthesized Mars environment. Camera movement types include displacements, up to 30 meters, at different altitudes (from 1,000 meters to 5,000 meters), and angular speed (up to 2.5 $^\circ/s$, in accordance to [70]), while image quality types include the injection of different levels of Gaussian noise, blur, brightness and contrast variations.

According to [70], the robustness has been evaluated exploiting two parameters: (i) *Number of Extracted Matches* (NEM), that identifies the number of matching points, and (ii) *Spatial Distribution of Points* (SDP), that measures how much the extracted matching points are uniformly spread in the image, defined as:

$$SDP = \frac{\sum_{i=1}^{N} -p_i \log p_i}{\log N} \tag{4.13}$$

where $p_i$ is computed as the number of matching points within an image cell (see Section 4.2.2.2) over the total number of extracted matching points in the frame, and $N$ is the number of image cells (i.e., 64).

Figure 4.22 shows the SDP results obtained from FEMIP [54] and SA-FEMIP by providing in input the images composing the aforementioned dataset. Thanks to the adaptive cell-based thresholding approach, the proposed architecture outperforms FEMIP results in every test case (i.e., Test Index). In particular, the improvements are very high (from Test Index 0 to 76) when the input images represent a landing zone characterized by few small rugged regions. This is visually highlighted in Figure 4.23 that depicts the matching points extracted by FEMIP (Figure 4.23(a)) and SA-FEMIP (Figure 4.23(b)). Each figure shows two consecutive input images with lines connecting the features that match in the two images.

Figure 4.24 shows the NEM versus different levels of injected Gaussian noise variance $\sigma_f^2$ (since FEMIP has a fixed $\sigma_f^2 = 2$, its NEM is represented by the dashed line).

A fixed $\sigma_f^2$ does not allow to reach the highest NEM for every noise level. Thus, exploiting the reconfigurable filter architecture it is possible to highly increase the number of extracted matches, as shown by the *Optimal* line in Figure 4.24. In order to follow the trend of this line,

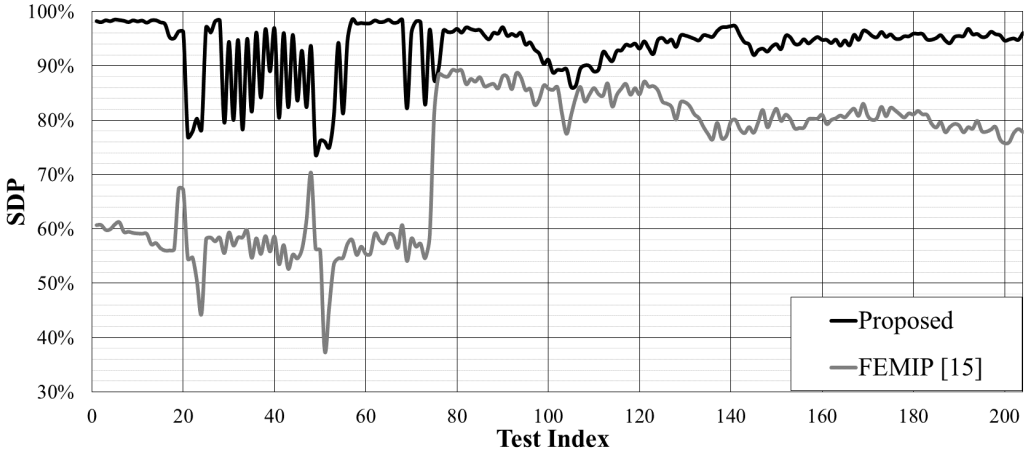


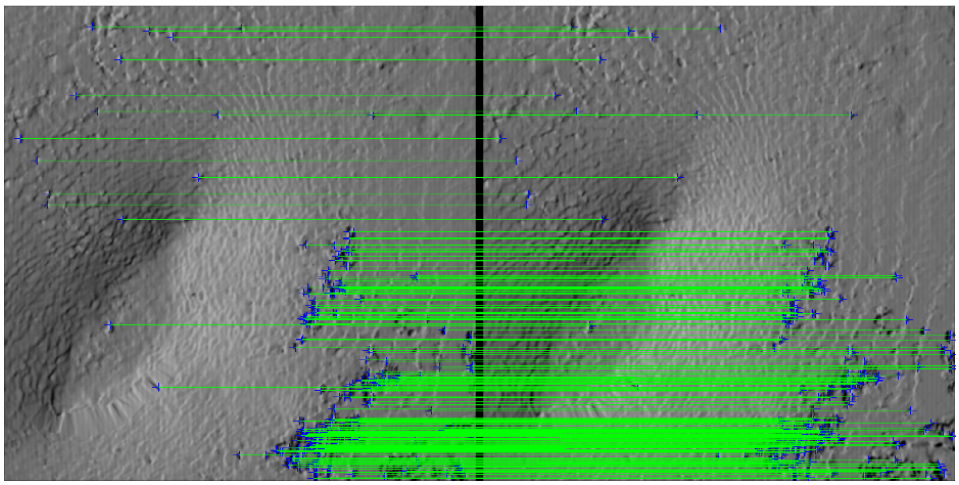

Figure 4.22: SDP results for FEMIP and the proposed architecture

in the proposed architecture 5 configurations for the RF module have been chosen. In particular, these configurations are associated to $\sigma_f^2$ equal to 0.5, 0.75, 1, 1.5 and 2, for the noise level ranges [0,100], [100,200], [200,300], [300,600], [600,1600], respectively. As can be seen in Figure 4.24, the usage of a reconfigurable filter increases the NEM value w.r.t. FEMIP up to 2 times, especially for a $\sigma_n^2$ lower than 600.

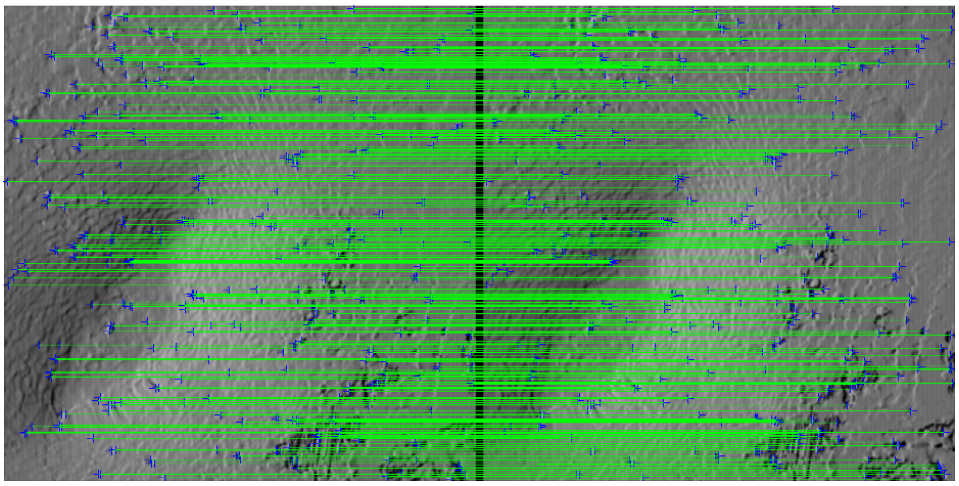
Moreover, as described in Section 4.2.2.1, the usage of the DPR enables to save resources with respect to use a static hardware architecture including 49 multipliers, each one with a multiplexer to select the right Gaussian kernel value. In the proposed architecture, using the same fixed-point data representation adopted in [54], the RM and the *Reconfiguration Controller* (Figure 4.12) require 5,320 LUTs and few registers. Instead, a static hardware architecture (as the one reported in [56]), with the same data parallelism, would require about 19,000 LUTs, leading to a save of 72% of hardware resources.

Since each bitstream for the RM module is 166 KB (for the selected FPGA device), to store the 5 configurations 830 KB are required in the external memory. Since the throughput of the *Reconfiguration Controller* is 400 MB/s (i.e., this value is limited by the maximum throughput of the *ICAP* [238]), the time required to reconfigure the RM is equal to 0.42 ms. This time fits the idle time of the external memory (i.e., $t_{idle}$ in Figure 4.21) that is equal to 1.2 ms (i.e., the time required by the *Matcher* to perform the *NMS* phase). For the sake of completeness, considering an operating frequency of SA-FEMIP chain equal to 60 MHz, the time required to perform the filtering and the matching tasks (i.e., $t_{filtering}$ and $t_{matching}$ in Figure 4.21) is 21.5 ms and 7.6 ms, respectively.

Eventually, Figure 4.25 shows the percentages of Correct Matches (CM) for the different filter configurations and injected noise levels. CM has been computed exploiting the knowledge about the camera movement between two consecutive images of the dataset. Starting from the



(a) FEMIP



(b) Proposed architecture

Figure 4.23: Example of extracted matches

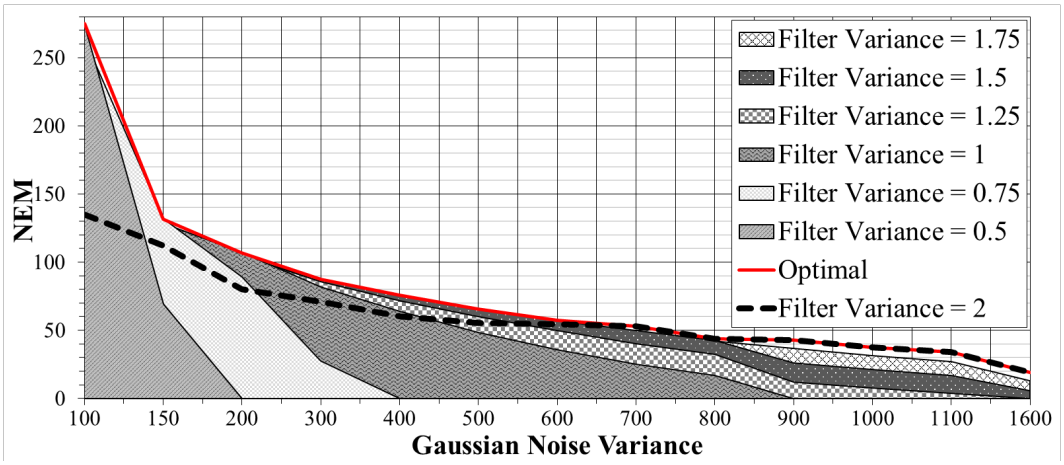



Figure 4.24: NEM results for different levels of injected Gaussian noise, varying the Gaussian Filter variance

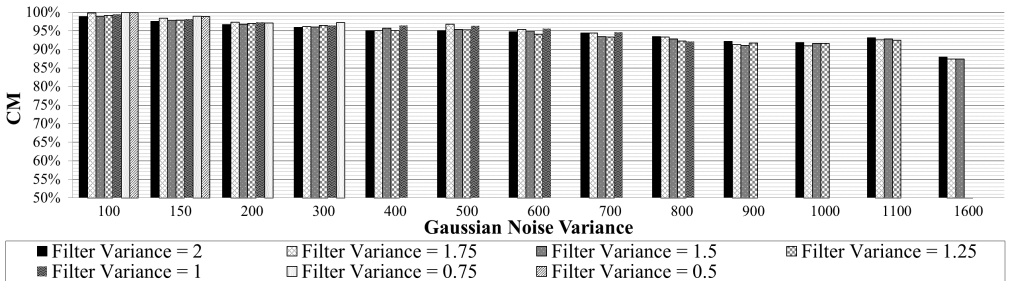



Figure 4.25: Correct Matches (CM) results for different levels of injected Gaussian noise, varying the Gaussian Filter variance

position of a matching point in the first image, it is possible to compute its expected position in the second image by using a three dimensional roto-traslation model. For each couple of images in the dataset this process has been automated through a *MATLAB* script. Then, the CM values have been computed by comparing the outputs of the script with the ones of the proposed architecture.

It is worth noting that, the CM values are not computed for every $\sigma_f^2$ since, as shown in Figure 4.24, with a filter characterized by a low variance it is not possible to extract matching points for very high noise levels.

As can be seen in Figure 4.25, the accuracy of the different filter configurations is higher than 85% for every noise level, and it is almost equal for a fixed noise level. These data demonstrate that the proposed filter is able to maximize the NEM, while preserving the correctness of its outputs.

CHAPTER 5

# ON ENHANCING DEPENDABILITY OF DYNAMIC PARTIAL RECONFIGURATION

In literature, DPR has been adopted in heterogeneous contexts, opening new scenarios on hardware/software co-design. It represents a promising solution to increase the overall SoC fault-tolerance [193] [55] [57], to enable on-demand hardware acceleration [180] [15] or, as also demonstrated in Chapter 4, to build self-adaptive systems [60]. Nevertheless, as discussed in Section 3.4, DPR comes at a cost of dependability issues related to the dynamic reconfiguration process itself.

This chapter discusses the proposed solutions for enhancing dynamic reconfiguration process dependability. In particular, it briefly recaps the issues related to run-time dynamic reconfiguration and the effect of mis-reconfigurations. Then, it presents two alternative ways to safely enhance reconfiguration process dependability. The former is essentially based on a set of rules to be applied at design-time, while the latter relies on the usage of a configurable hardware self-reconfiguration manager that must be instantiated within the target system. Both approaches can be employed to safely enable self-adaptivity mechanisms in the designed systems without decreasing the dependability levels required by the target applications.

## 5.1 Dependability issues in DPR

As mentioned in Section 3.4, *Xilinx*'s EDA tools generate a separate configuration file (called *partial bitstream* or *partial bit-file*) for each module to be mapped into a specific reconfigurable region. The partial reconfiguration process can be then activated at run-time by loading a partial bitstream inside the FPGA through a dedicated configuration port, that can be either external (e.g., *SelectMAP* or internal (i.e., ICAP) [238]. ICAP is usually preferred when building self-adaptive systems because it does not require external hardware controllers in charge of managing

configuration. However, if the design does not include a soft/hard-core microprocessor embedded in the FPGA, such as the Microblaze [223] or PowerPC [224], a user-designed reconfiguration manager, placed in the static portion of the design, is required to load the partial bitstream through the internal configuration port.

As mentioned in Section 3.4, in general, the routing resources inside a reconfigurable area can be also exploited for the intercommunication among static modules. In the DPR user guide [231], Xilinx reports that, when a partial reconfiguration design is placed and routed, *"the static routes can route through reconfigurable areas"*. Instead, *"routes within reconfigurable modules cannot route outside the boundaries of the user-defined reconfigurable partition"*.

Let us suppose that the *partial bitstream* is stored in an off-chip memory (RAM or Flash), in addition to the hypothesis of fault-free internal logic. If a corrupted bitstream is loaded into the FPGA, errors can be localized in the header section or the data section of the bitstream. If the header is corrupted, the static portion of the design could be damaged, thus requiring an overall FPGA reconfiguration. The time overhead caused by the full reconfiguration may be unacceptable in time-critical applications such as hard real-time scenarios. The full reprogramming may also impact high-dependable applications due to the disruptive consequences on the static portion of the design.

On the contrary, if errors are localized in the data part of the bitstream, a faulty reconfigured module will be instantiated. To restore a correct FPGA configuration, a DPR in the same reconfigurable area is sufficient. However, if links between static modules are routed through the reconfigurable area, a global reset of the FPGA could be required. For instance, if interconnections between the Reconfiguration Manager (RM) and the Memory Controller (MC) cross the reconfigurable area, a faulty DPR could damage these links, isolating the RM. The RM would therefore be unable to communicate with MC to read the partial bitstreams, therefore preventing any repair mechanism.

## 5.2 Dependable DPR with minimal area and time overheads

This section discusses a methodology aimed at providing a dependable DPR flow, minimizing both area occupation and reconfiguration time. Exploiting data provided by Xilinx manuals, a tool has been developed to isolate and protect the most critical sections of a *partial bitstream*. In addition, design rules are also provided in order to protect the most critical modules in the static portion of the FPGA. A System-on-Chip (SoC) use case will be analyzed to demonstrate the feasibility and effectiveness of such methodologies on complex designs.

The problem of dependable partial reconfiguration has been addressed by *Xilinx* in the *Partial Reconfiguration User Guide* [231], where a partial bitstream Cyclic Redundancy Code (CRC) checking is suggested.

The main idea is to split the original partial bit file into *blocks*, and for each block to calculate a single word CRC signature. Finally, the partial bit file is reassembled, combining the blocks and their corresponding CRC signatures (Figure 5.1).

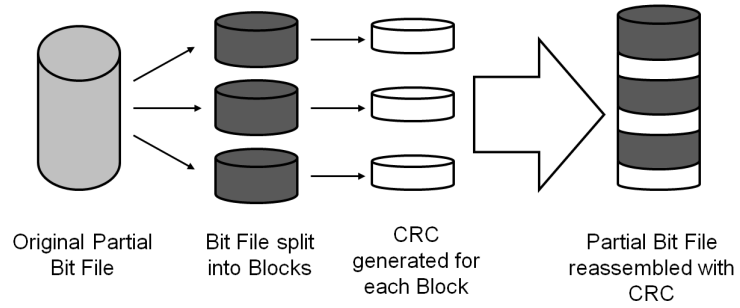



Figure 5.1: Bitstream generation [229].

At reconfiguration time, whenever a new block is loaded from the external memory, the CRC signature is recalculated by a dedicated hardware component, and the block is stored into a *BRAM* inside the FPGA.

When the whole block has been transferred from the external memory, the reconfiguration manager compares the received CRC signature with the calculated one (Fig. 5.2).

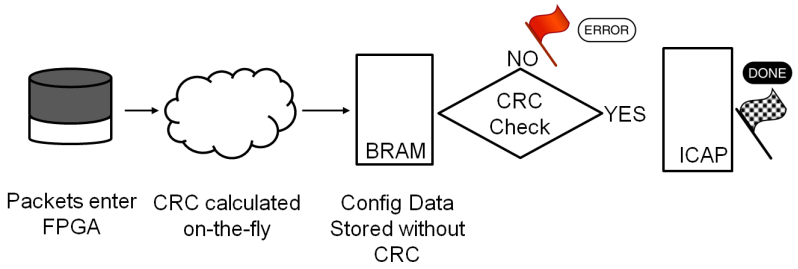



Figure 5.2: Bitstream Loading process [229].

If the CRC comparison is successful, the block can be sent to the ICAP and the related portion of the frame reconfigured. Otherwise, the reconfiguration manager must reload the block from the external memory.

This implementation requires a CRC evaluator, a Finite State Machine (FSM) for the control of the reconfiguration process (i.e., a Reconfiguration Manager) and a set of embedded memories to buffer the data.

Depending on the available FPGA resources, the designer need to choose the best block size, that impacts on both reconfiguration time and internal memory occupation.

On the one hand, when increasing the number $N$ of blocks (i.e., when decreasing the number of words per block), a higher number of CRC signatures must be stored, leading to an increase

of bitstream size. At the same time, the memory occupation becomes smaller, since the required buffering capability equals the block dimension.

On the other hand, when decreasing the number of blocks, fewer CRC signatures must be stored in the bitstream, thus decreasing the total amount of words in the external memory. However, during the CRC checking process, more words must be stored in the memory embedded in the FPGA, thus increasing the area occupation.

The total reconfiguration time is due to two contributions:

$$T_{X-CRC} = T_{read} + T_{block} \tag{5.1}$$

where:

- $T_{read}$ is the time required to load the bit file from the external memory:

$$T_{read} = \frac{K+N}{\min(f_{ICAP}, f_{Mem})} \tag{5.2}$$

  where $K$ is the bitstream dimension in terms of 32 bit words; $N$ is the number of CRC signatures in terms of 32 bit words; $f_{ICAP}$ is the working frequency of the ICAP; $f_{Mem}$ is the memory working frequency.

- $T_{block}$ is the time spent loading the buffered block from BRAM to the ICAP:

$$T_{block} = \frac{K/N}{f_{ICAP}} \tag{5.3}$$

  where $K/N$ is the block dimension in terms of 32 bit words.

Using this model, it is possible to evaluate how the reconfiguration time is influenced by the block dimension. Fig. 5.3 plots the reconfiguration time as a function of the block dimension $K/N$. This evaluation has been performed with different bitstream file dimensions (# frames involved in the reconfiguration).

Fig. 5.3 shows the reconfiguration time considering a 100 MHz working frequency for both ICAP and external memory. Note that, with a block dimension of a single word ($N$=$K$), one CRC signature for every word is required. In this case, $T_{read}$ becomes the most relevant term, but no memory is required for buffering.

The same time overhead occurs when a block is as long as the bitstream ($N$=1). While there is just one *CRC* signature, the whole bistream must be buffered before being sent to the ICAP, resulting in high embedded memory occupation. This significantly increases $T_{block}$.

### 5.2.1 Proposed Methodology and Design Rules

Despite the discussed solution is fairly comprehensive, it may implies significant time and area overheads. In the sequel an alternative methodology will be presented and analyzed. The pro-

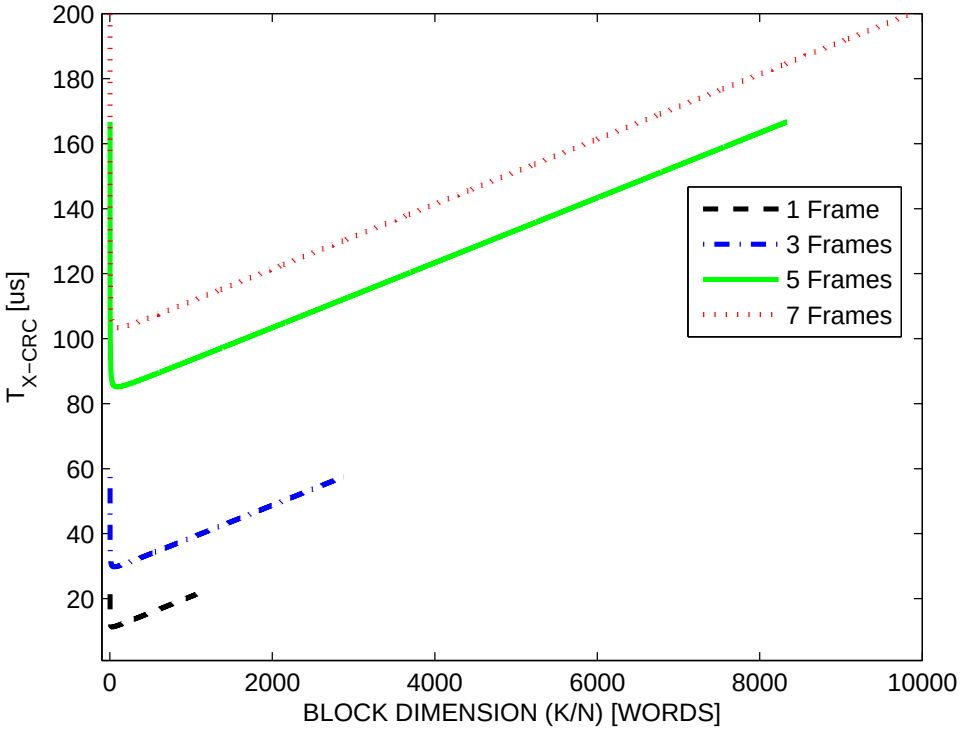



Figure 5.3: Reconfiguration time of Xilinx solution

posed methodology tries to reduce these overheads by protecting just the critical part of the partial bitstream. Some ad-hoc design rules are also introduced.

#### 5.2.1.1 Partial bitstream file splitting

As mentioned and explained in Sec. 3.4.1, the partial bit-file is composed of three main parts. The first part contains frame addressing and control information. The second includes the data for the reconfiguration in the selected frames. The last part includes the built-in ICAP CRC checksum.

It is straightforward that, in terms of dependability, the most critical part is the first one, since it defines the portion of the FPGA to reconfigure. In fact, if an error occurs in an address or control information, a static portion of the FPGA could be unintentionally reconfigured and the system could become inoperative.

The proposed approach deeply protects, with CRC signatures, this portion of the partial bitstream that contains the most critical words, letting the rest unprotected (see Figure 5.4).

At reconfiguration time, whenever a critical word is read from the external memory, it is fol-

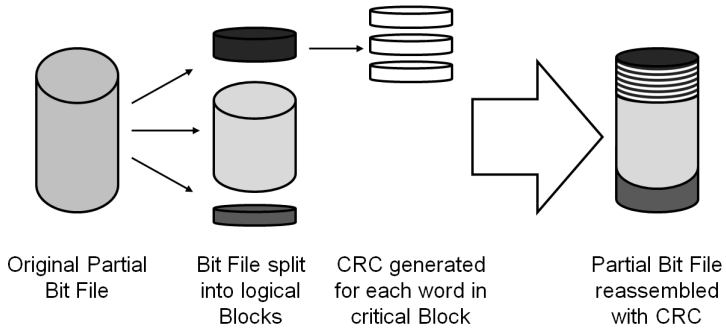



Figure 5.4: Bitstream generation with the proposed solution.

lowed by the relative CRC signature. The critical word will be temporarily stored and the relative CRC signature is calculated on-the-fly. The next word read out from external memory will be the software-calculated CRC: it will be compared with the hardware computed and, if they are equal, the critical word will be sent to the ICAP. The non-critical words are loaded from the external memory and directly sent to the ICAP, without any time overhead or buffering (see Figure 5.5).

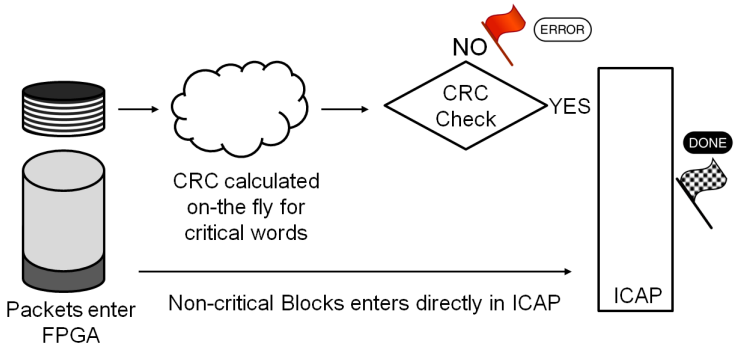



Figure 5.5: Bitstream loading process with the proposed solution.

Analyzing more in detail the proposed solution, it requires, on the software side, a script that is able to split the bitstream depending on the semantic and to generate CRC signatures for critical words. In hardware a *CRC* calculator and a FSM for the control of the reconfiguration process have to be implemented, while no FPGA embedded memory block is required.

The reconfiguration time with the proposed CRC checking is

$$T_{OUR-CRC} = \frac{K + C}{\min(f_{ICAP}, f_{Mem})} \tag{5.4}$$

where $K$ is the bitstream dimension in terms of 32 bit words; $C$ is the number of critical words; $f_{ICAP}$ is the working frequency of the ICAP; $f_{Mem}$ is the memory working frequency.

Fig. 5.6 plots the ratio $T_{X-CRC}/T_{OUR-CRC}$, which proved to be always >1.

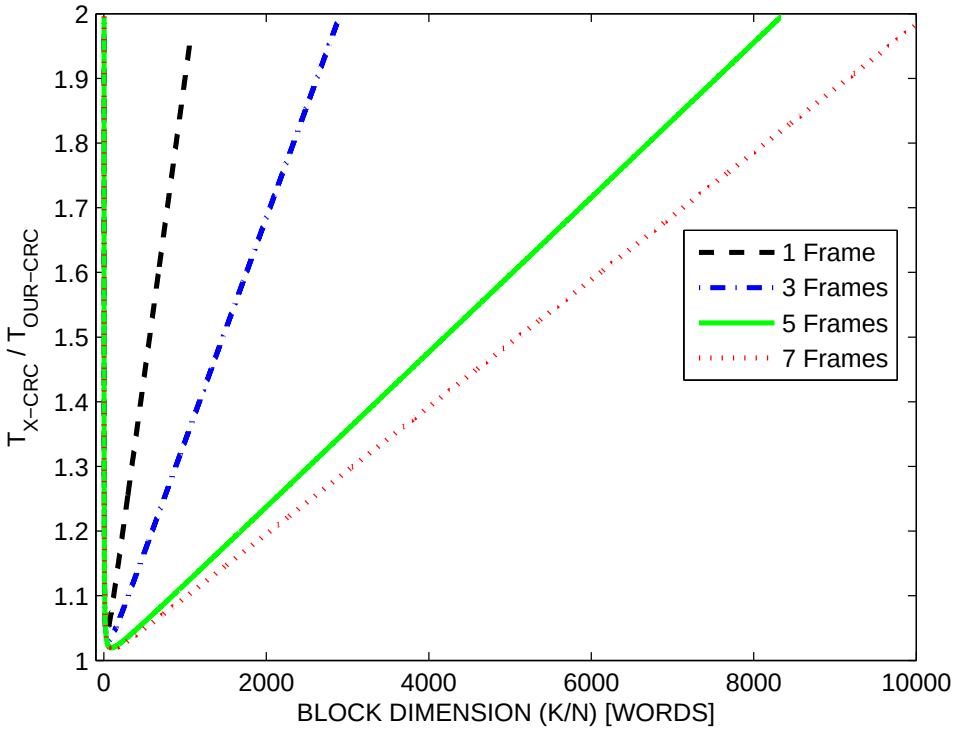



Figure 5.6: Comparison between proposed solution and Xilinx solution

Despite the proved time and area occupation advantages of the proposed solution, to assure a high dependability of the reconfiguration process, we have to guarantee that an error in the non-checked part of the bitstream file will not lead to a fault in the system. The jeopardy is that some static connections are routed in the reconfigurable area, and, due to a faulty reconfiguration process, the link between two points could be broken. This goal is achieved by fulfilling the following rules:

1. potential critical links must not cross any reconfigurable area
2. connection inside critical modules must not cross (i.e., be routed through) reconfigurable areas.

#### 5.2.1.2 Critical links protection

To ensure a dependable reconfiguration process, critical connections must be protected. These include:

- External-Memory to Memory Controller links;

- Memory Controller to Reconfiguration Manager links;
- Reconfiguration Manager to ICAP links.

In order to guarantee that these links do not cross the reconfigurable area, after the automatic routing performed by the synthesis tool, the layout must be checked and some links manually re-routed, if required.

#### 5.2.1.3 Critical modules protection

The second rule imposes that all critical modules must be protected. In systems which use partial reconfiguration, since bitstreams are loaded from the external memory, all modules involved in the communication between the external memory and the ICAP must be considered critical (see Fig. 5.7). In addition, also design specific modules could be considered critical. Their integrity can be preserved by constraining critical modules in predefined physical region called *partitions*. Electronic Design Automation (EDA) tools enables the user to manually place a module in a specific area, guaranteeing that all the specified hardware and the related connections are inside the physical regions.

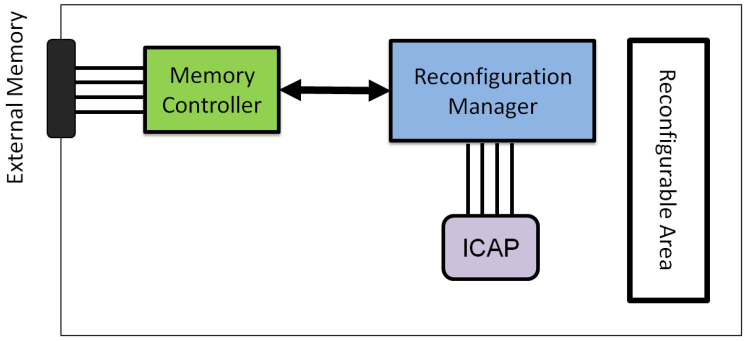



Figure 5.7: Critical connections and cores

### 5.2.2 Experimental results

This section reports a set of experiments performed to validate the proposed methodology and to compare its performance with the reference solution (Figures 5.1 and 5.2).

The experimental setup includes a *Leon3* processor-based SoC [45, 83], implemented on a *Xilinx ML403* demo board, equipped with a *Xilinx Virtex 4* FPGA device and 64 MB of DDR SDRAM [218]. The SoC design includes a reconfigurable area and an ad-hoc reconfiguration manager connected to the AMBA bus and an internal timer able to measure the partial reconfiguration time. The reconfiguration manager is able to:

- address the external memory, loading bitstream files without any CPU intervention (*Direct-Memory-Access*);
- perform DPR through ICAP;
- perform the CRC check;
- automatically manage the whole reconfiguration process, even in presence of errors.

The *Leon3* processor works at 66 MHz, the ICAP controller and, the DDR SDRAM at 100 MHz.

#### 5.2.2.1 Reference solution implementation

The reference solution has been implemented employing a parallel CRC-32 computation module to minimize the CRC latency. The selected polynomial is *0x90022004* which guarantees Hamming distance equal to 6 [113].

For each considered reconfigurable module a partial bit-file of 1,969 32-bit words has been generated.

We evaluated the reconfiguration time and the required area considering the following block sizes: 4, 16, 32, 44, 64, 128, 256, 512 32-bit words. Fig. 5.8 shows the relation between the reconfiguration time and the block size. The solid line plots the reconfiguration time calculated using Equation 5.1 while the dots report the measured reconfiguration time for the 8 considered block sizes. The graph confirms that the considered mathematical model provides a good estimate of the configuration time and can be used to identify the best block size for a given design.

#### 5.2.2.2 Proposed approach implementation

Differently from the previous solution, the proposed approach is designed to protect 16-bit critical words. A smaller CRC can therefore be adopted. We implemented a parallel CRC-16 with polynomial equal to *0x968B* which guarantees Hamming distance equal to 7 [115].

The presented design rules have been applied to the proposed design. Partitions have been created using *Xilinx* EDA tools to protect the SoC critical modules (i.e., Reconfiguration Manager (RM) and Memory Controller (MC)). To ensure that the processor keeps running also after a faulty reconfiguration, the *Leon3* has been constrained in a specific region, also (see Figure 5.9). This introduces a minimal degradation (1.2%) in the maximum working frequency.

Finally, all critical connections have been checked, in order to assure that they do not cross the reconfigurable area, and only 2 links were manually re-routed.

#### 5.2.2.3 Comparison

Table 5.1 compares the two analyzed solutions, in terms of area occupation and fault free reconfiguration time. The assets of the proposed solution are no BRAM occupation and a shorter

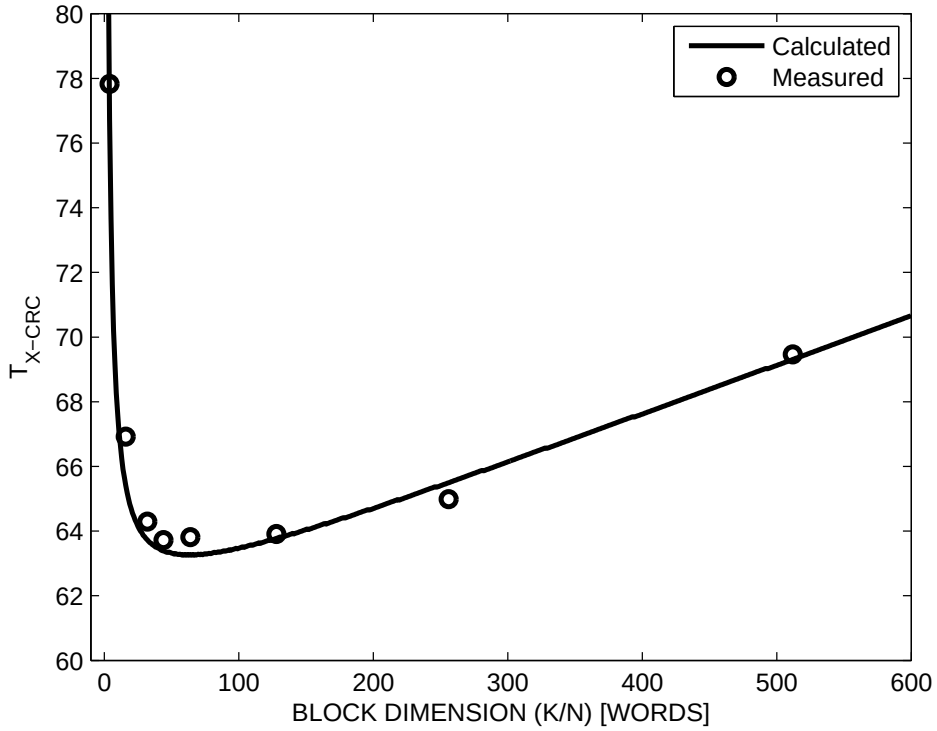



Figure 5.8: Reconfiguration time with 2 Frames

Table 5.1: Area occupation and reconfiguration time of different implementations

| **Solution** | **Block Size [bit]** | **CRC [#]** | **BRAM [#]** | **RM [# slices]** | **Reconfig. Time [$us$]** |
|---|---|---|---|---|---|
| w/o CRC | 0 | 0 | 0 | 197 (3.60%) | 61.04 |
| Proposed Approach | 1x16 | 42 | 0 | 295 (5.39%) | 61.78 |
| Ref. Solution | 64 x 32 | 31 | 1 | 290 (5.30%) | 63.72 |
| Ref. Solution | 1 x 32 | 1,969 | 0 | 290 (5.30%) | 77.88 |
| Ref. Solution | 1,969 x 32 | 1 | 8 | 290 (5.30%) | 73.49 |

reconfiguration time compared to the Xilinx solution, with a very small overhead in the configuration manager (increase of 1.7% of slices) due to a more complex FSM.For sake of completeness, Table 5.1 also provides information about the worst and best cases for the reference solution, and a CRC free DPR system.

So far the performance comparison has considered the fault free DPR time, only. The rest of this section will compare DPR performance in case of faults in the bitstream, which have been injected in the words of the data portion, only. This condition is conservative, since it represents

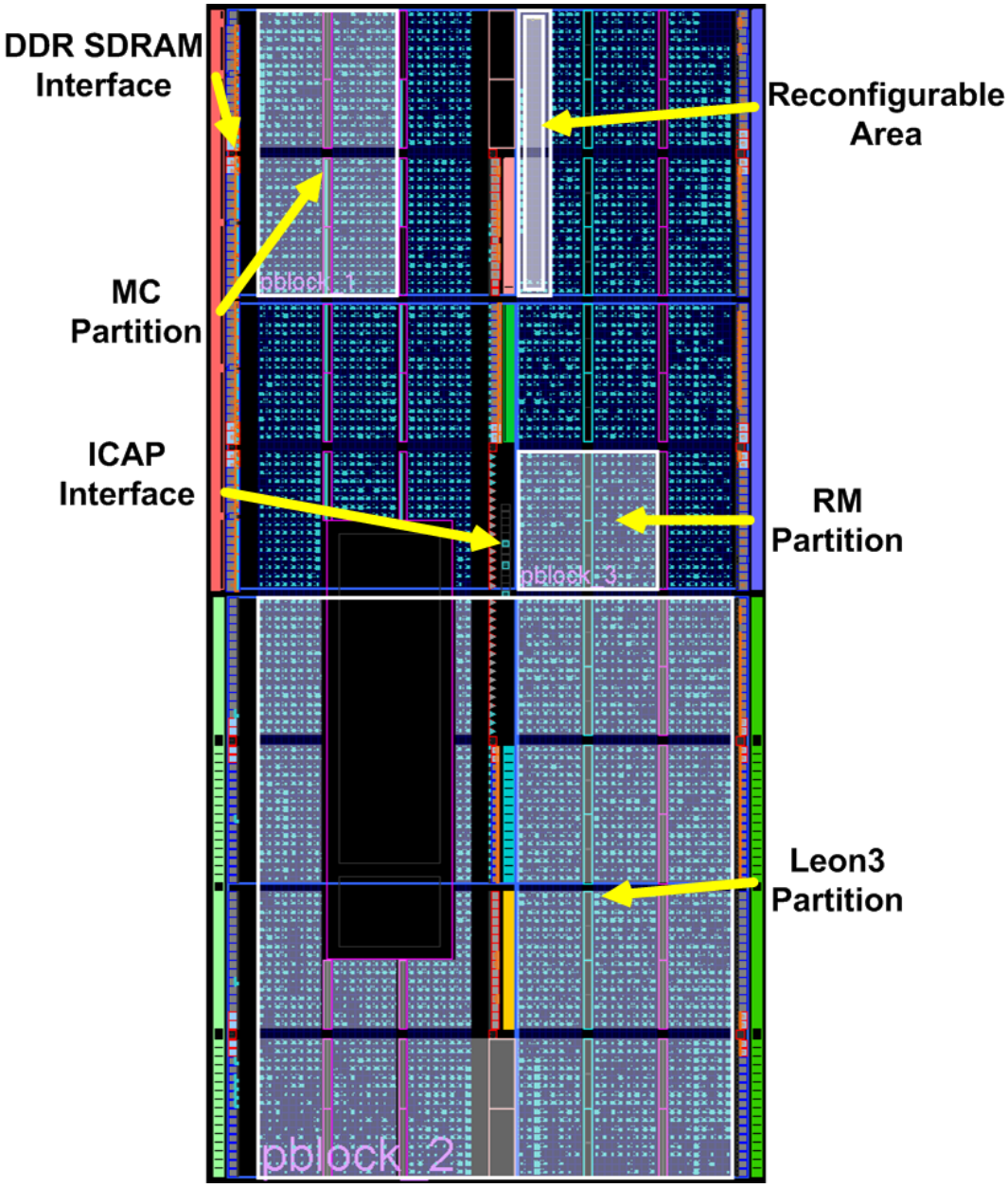



Figure 5.9: *Xilinx PlanAhead* tool device view

the worst-case condition for the proposed solution. In fact, when applying the reference solution, if a faulty block is loaded, the error is detected as soon as the CRC of the block is checked. The block can be immediately reloaded, thus introducing a time overhead equal to the block loading time. In the proposed solution, if an error occurs in a critical word, it is immediately detected

enabling the system to reload the corrupted word. On the other hand, if the error occurs in the data portion, it will be detected only at the end of the reconfiguration process, during the ICAP CRC check. In this case the full DPR process must be restarted since the reconfigurable area has been corrupted, thus introducing a higher overall reconfiguration time overhead.

The time overhead introduced by errors in the loaded blocks for the two considered solutions is therefore influenced by the DPR rate, and by the word error probability observed when loading bitstream blocks. Figure 5.10 and Figure 5.11 analyze the difference in system activity time spent for DPR in the two solutions over a one day observation period for different DPR rates and word error probabilities.

Figure 5.10 analyzes the case of a partial bit file composed of 1,969 32-bit words.

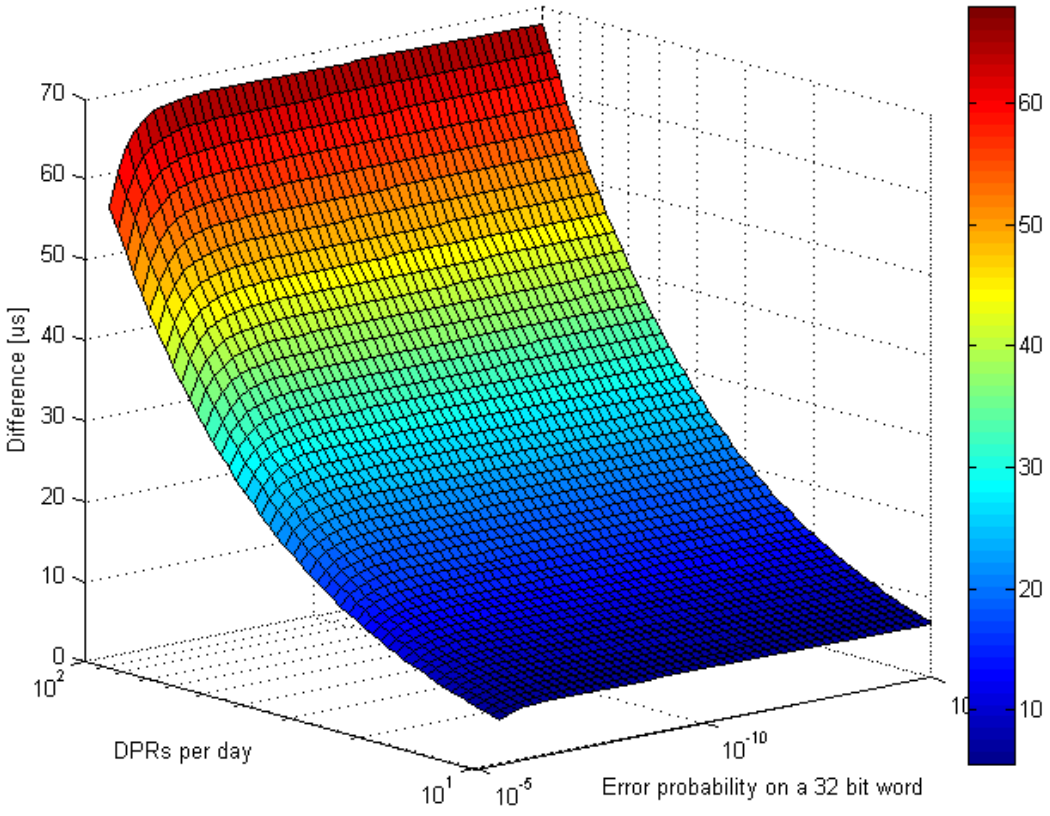



Figure 5.10: Difference of DPRs time in 1 day - Bitstream size equal to 1,969 32-bit words.

Figure 5.11 performs a similar analysis, but considering a larger reconfigurable area composed of 8 frames, i.e., a partial bit file of 11,040 32-bit words.

In last case, when the word error probability increases over $10^{-6}$, the reference solution should be preferred. This is due to the additional reconfiguration process required in the proposed so-

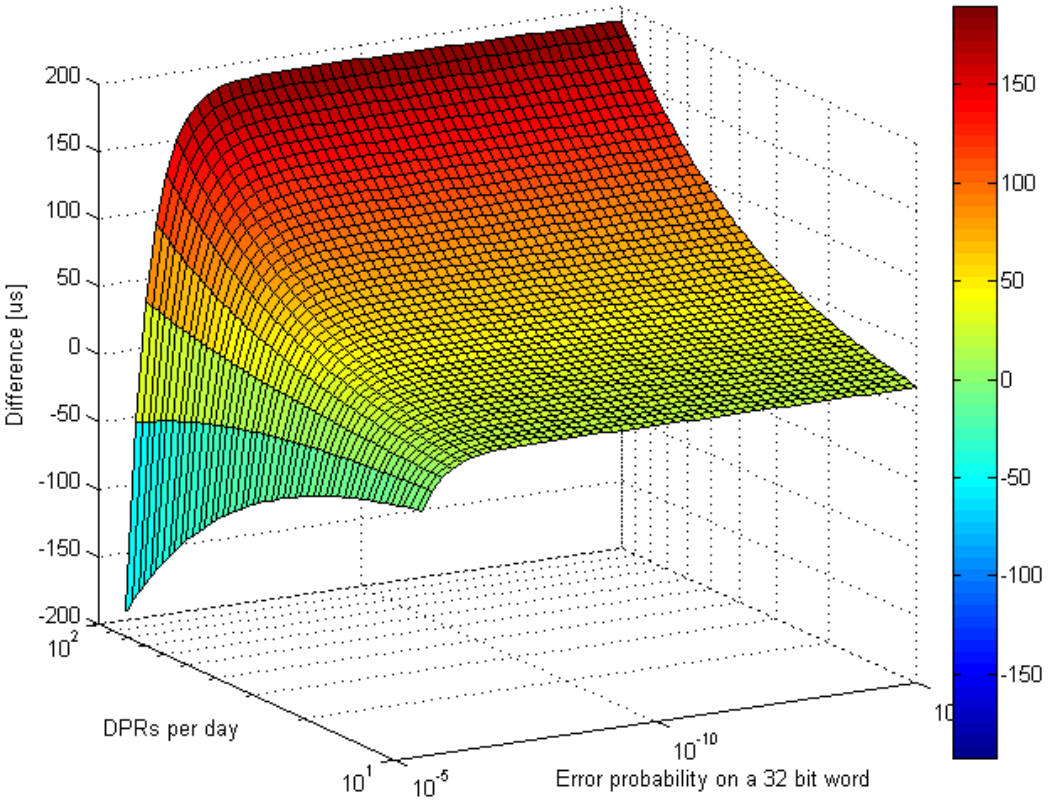



Figure 5.11: Difference of DPRs time in 1 day - Bitstream size equal to 11,040 32-bit words.

lution whenever configuration data words are corrupted.

## 5.3 A portable open-source controller for safe Dynamic Partial Reconfiguration

Although DPR can be used to increase reliability figures of a system, its adoption in applications demanding high reliability is actually very limited for two main reasons. The first one concerns the additional complexity introduced during the system design phase. In fact, to efficiently enable and manage run-time DPR, designers are often required to develop ad-hoc external or embedded hardware controllers. The latter, instead, is related to the dependability of the reconfiguration process itself (see Sections 3.5.1 and 5.1). DPR exposes the system to errors affecting both the hardware controller and the *bitstreams* that are used to overwrite portions of the FPGA configuration memory content at run-time. These errors are very critical since a mis-reconfiguration can lead, in the worst case, to a permanent disruption of the entire system functionality. Recovering from such errors could require a full device reconfiguration and/or reset of system operations.

To tackle the aforementioned issues this section presents a portable open-source embedded controller for safe dynamic and partial reconfiguration of systems implemented on *Xilinx* FPGAs. The main novelties with respect to the state-of-the-art solutions mainly concern its high configurability and the possibility to introduce embedded error detection and correction circuitry, which monitors for bitstreams data errors during reconfigurations. Schemes for increasing the reliability of the proposed controller with respect to errors affecting the FPGA device are also discussed. Its flexibility allows designers (i) to instantiate the controller as a peripheral in processor-based systems or in conjunction with custom FSMs, and (ii) to tune its hardware and reconfiguration timing overheads based on the requirements of the target application.

It is worth to mention that the HDL source code has been made available through the open source *Cobham Gaisler GRLIB GPL* IP-cores library [45].

### 5.3.1 Related Works

Xilinx provides several IP-cores for enabling DPR and interfacing user designs with the ICAP. *XPS HWICAP* [227] and *AXI HWICAP* [235] represent two DPR controllers equipped with PLB and AXI4-Lite slave bus interfaces, respectively. Their main limitations concern the restricted applicability to processor-based systems and the inefficiency of data transfers due to the slave interface, that leads to reconfiguration throughputs far below the ICAP theoretical limit (i.e., $3.2Gbps$). A slightly more flexible solution is represented by the PRC/EPRC controller [232], which provides a FIFO user interface that allows its usage with custom user logic. However, its source code is not provided and netlists are available only for *Virtex-5* and *Virtex-6* devices.

Several works have been proposed in literature to overcome the limitations of the IP-cores provided by *Xilinx*. They mainly focus on maximizing reconfiguration throughput by designing DPR controllers that best fit the target system and use model [43, 48, 95, 131, 168]. Recent examples can be found in [212] and [74]. In [212], authors improve *AXI HWICAP* by adding an AXI4 DMA interface to reconfigure a Zynq device. Their solution reaches a throughput of 382 MBytes/sec. In [74], an hardware manager with FAT16 file system support is proposed, reaching 398.6 MBytes/sec while imposing a bitstream size limit.

In some cases, the approaches proposed in literature rely on ICAP overclocking in order to accelerate the reconfiguration throughput beyond the *Xilinx* specifications [26, 98, 103, 171]. Although these methods provide a dramatically improved reconfiguration speed, they are not suitable when targeting a safe DPR, since process, voltage, or temperature variations may lead to malfunctions if ICAP specifications are not fulfilled [103].

Recently, generic DPR controllers have been proposed in [211] and [199]. In particular, in [211] authors present a reconfiguration manager, implemented on a *Virtex-6* device, which approaches the maximum reconfiguration throughput when the bitstream is read from a DDR3 memory, while [199] proposes a controller that achieves roughly 253 MB/sec when directly inter-

faced to an external SD Flash memory.

None of the aforementioned solutions takes into account DPR dependability issues. Few works can be found in literature targeting the development of reliable DPR controllers aimed at increasing overall reconfiguration process reliability. On one hand [101] proposes a reliable ICAP controller targeting only FPGA configuration memory scrubbing while, on the other hand, in [72] and [71] authors propose and discuss several alternatives for increasing reliability of embedded DPR controllers (e.g., by applying Triple or Dual Modular Redundancy). Although the proposed solutions provide DPR controllers that are robust w.r.t. faults affecting the FPGA device, they do not tackle bitstream integrity issues.

The proposed portable and open-source controller, instead, tries to provide a comprehensive solution for enabling safe dynamic and partial reconfiguration. The flexibility of the proposed controller allows designers to adopt it in applications requiring, as example, reconfigurable module relocation [67] or blind FPGA configuration memory scrubbing [102].

### 5.3.2 Proposed architecture

The proposed DPR controller has been designed in order to provide portability and configurability on different FPGA families. In particular, it can be configured at design-time through VHDL generics that allow to select the target device and define the operation performed by the controller at run-time. Depending on the target system and application requirements it can be configured to operate in four different modes:

- Synchronous DPR;
- Asynchronous DPR;
- Dependable DPR with Cyclic Redundancy Check ($D^2$*PR-CRC*);
- Dependable DPR with Error Detection and Correction ($D^2$*PR-EDAC*);

The following subsections detail the architectures of the proposed controller for each operating mode, discussing when a particular configuration should be preferred to implement a safe DPR.

#### 5.3.2.1 Synchronous/Asynchronous DPR

Figure 5.12 shows the architecture of the proposed controller in its basic configuration, i.e., *Synchronous* mode.

It mainly consists of a FSM that drives and monitor the ICAP, supported by a Direct Memory Access (DMA) engine and a control and status registers block. All modules, including the ICAP, operate synchronously with respect to the input system clock.

The control and status registers block includes several registers that can be read and written to setup, trigger, and monitor the reconfiguration process at run-time. The DMA is in charge of

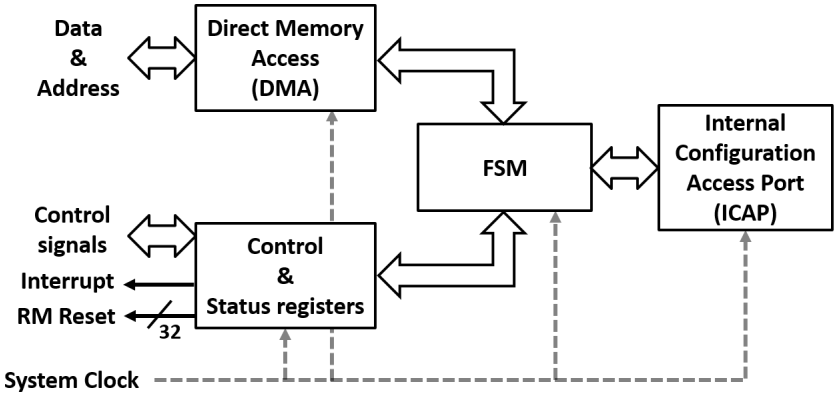



Figure 5.12: DPR controller architecture for Synchronous DPR mode.

retrieving 32-bit bitstream data words from an embedded or external memory, starting from the address specified by the user through a 32-bit register in the Control and Status registers block. Obviously, depending on the actual system implementation, a custom wrapper may be needed to adapt the interface of the DMA engine to the user requirements (e.g., to connect the controller to a bus infrastructure or directly to a memory controller. An *interrupt* signal is asserted whenever an error is encountered or to report the end of the reconfiguration process.

An additional functionality is provided by the 32-bit *RM Reset* output signal. At run-time, whenever a reconfigurable module is replaced, the logic inside that module must be reset before being activated with the new reconfigured functionality [250]. Each bit of the *RM Reset* signal acts as synchronous reset for each reconfigurable module in the design. It is automatically asserted by the DPR controller at the end of the reconfiguration process in order to initialize only the actually reconfigured logic. This functionality can be enabled or disabled through a 32-bit register for up to 32 different reconfigurable regions of the FPGA.

If the system clock frequency is greater than the one sustainable by the ICAP (i.e., $100Mhz$), the proposed controller can be configured in *Asynchronous DPR* mode. As shown in Figure 5.13, the *Synchronous* architecture (see Fig. 5.12) is extended by adding two control units, a FIFO buffer, and clock generation circuitry.

The FIFO buffer is implemented using one or more FPGA embedded Block-RAMs configured in 512x36 or 1024x36 bit mode depending on the target device [244], and its depth can be configured through VHDL generics. This buffer is used to transfer data across two different clock domains, i.e., *System clock* and *ICAP clock*. *ICAP clock* is generated exploiting a clock manager hard macro [244] sourced by the *System clock*. The *FSM READ* module is in charge of managing the bitstream data retrieval through the DMA. Contrary to the *Synchronous* mode, data are written into the *FIFO* buffer and not directly into the ICAP. As soon as data are available in the buffer, the *FSM WRITE* controller reads-out and deliver them to the ICAP.

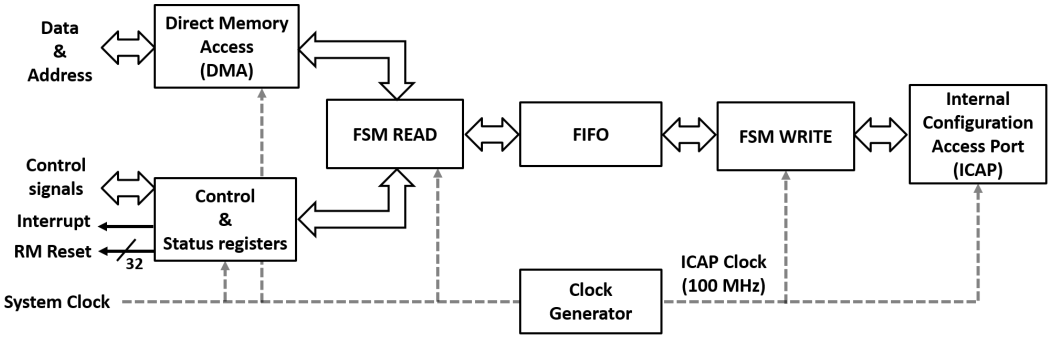



Figure 5.13: DPR controller architecture for Asynchronous DPR mode.

The registers block, interrupt and DMA engine make the proposed architecture suitable to be employed in systems with custom user interfaces and also in processor-based bus infrastructures, allowing the processor to control and manage the reconfiguration process through software drivers. In this last case, the advantage deriving from the adoption of DMA and interrupt is twofold: on one hand it frees the processor from directly managing the data transfer and repeatedly polling the status of the reconfiguration, while, on the other hand it accelerates bitstream data retrieval operations [131].

The maximum reconfiguration throughput in both operating modes can be estimated by the following equation:

$$Max\_TP = \frac{BS}{max\{T_{sys}, T_{ICAP}\} \cdot \frac{BS}{32}} \leq 400MBytes/s \tag{5.5}$$

where $BS$ represents the bitstream size in bits, while $T_{sys}$ and $T_{ICAP}$ are the system and ICAP clock periods, respectively. In Equation 5.5, the denominator represents the time required to write $BS$ bits of data to the 32-bit interface of the ICAP.

Practically, the reconfiguration throughput depends on the actual system implementation. When retrieving bitstream data, communication overheads due to the adopted bus protocol or memory latencies must be taken into account. Therefore, the actual throughput that can be achieved when instatiating the proposed controller in the target system can be roughly estimated by the following equation:

$$TP = \min\{Max\_TP, BW_{BUS}, BW_{memory}\} \tag{5.6}$$

where $BW_{BUS}$ represents the actual bandwidth of the bus infrastructure, while $BW_{memory}$ is the bandwidth provided by the memory controller used to interface the embedded or external memory storing bitstream data.

Both *Synchronous* and *Asynchronous* configurations do not natively provide any bitstream error detection and/or correction functionality. They represent two alternative solutions to en-

able DPR with very low hardware overheads. These configurations can be used when reliability is not a major concern, or if the user logic or memory controller used to retrieve bitstream data embed error detection and correction capabilities. Unless a radiation hardened FPGA (e.g, *Virtex5-QV* [247]) is employed, the proposed controller can be effectively protected against faults affecting the FPGA device by implementing Dual or Triple Modular Redundancy (TMR), or one of the schemes proposed in [71]. However, when used in *Asynchronous* mode, the controller includes one or more embedded Block-RAMs used to implement the FIFO buffer. Applying dual or triple modular redundancy will also double or triple the memory resources needed to implement the proposed DPR controller. Embedded memories often represent a critical resource when designing complex systems. To limit this overhead an alternative scheme is proposed and depicted in Figure 5.14, where TMR is applied to all the modules composing the DPR controller, while the FIFO buffer is not replicated.

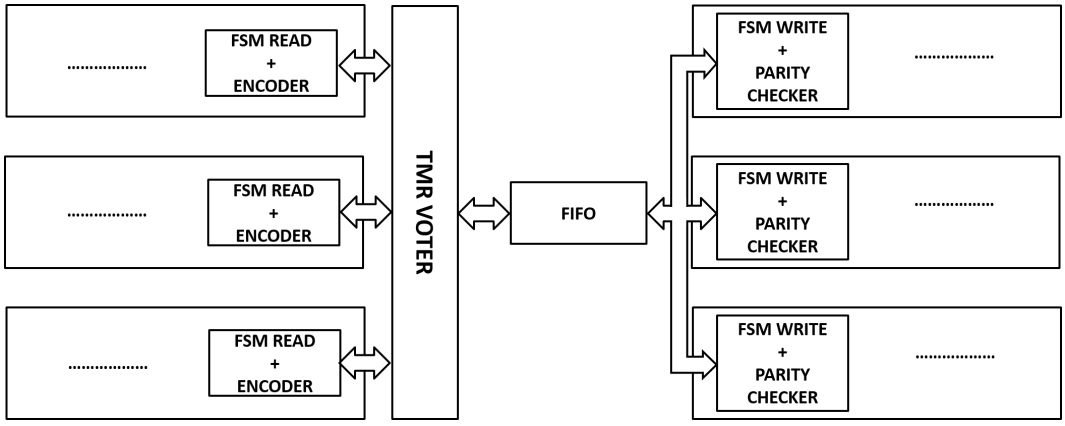



Figure 5.14: Proposed TMR approach applied to the Asynchronous DPR mode architecture.

Taking into account that the bitstream data word length is 32-bits and embedded Block-RAMs are configured to host 36 bit words, each memory line presents 4 unused bits. These additional bits can be employed to implement error detection through an *even* or *odd* parity scheme [82], providing very low hardware overhead. In particular, *FSM READ* and *FSM WRITE* include additional logic for encoding and decoding bitstream data. Parity bits are computed on the input data before being written by the *FSM READ* into the FIFO. The incoming 32-bit bitstream word is split in 4 8-bits sub-words. A *even* or *odd* parity bit is computed for each sub-word, leading to an encoded 36-bit word written in the FIFO. During buffer read-out, performed by *FSM WRITE*, parity bits are checked and if no errors are detected the extracted 32-bit word is written into the ICAP. This simple parity scheme provides detection of single or an odd number of errors within each 8-bits sub-word stored in the FIFO buffer. It represents a realistic countermeasure since usually physical memory layout methodologies are employed to interleave memory cells, thus minimizing the occurrences of multiple errors in a single memory word [219]. As will be demonstrated

in Section 5.3.3, since both encoder and parity checker mainly consists of simple *xor* trees [82], the proposed methodology provides low hardware overhead, in terms of logic resources, while reducing memory requirements with respect to a full TMR approach.

#### 5.3.2.2 Dependable DPR ($D^2PR$)

As mentioned in Section 5.1, monitoring for partial bitstream data errors is essential to avoid FPGA mis-reconfigurations due to a corrupted partial bitstream. Two alternative methods and DPR controller architectures are presented. Both methodologies are based on introducing information redundancy at design-time in the partial bitstream data files generated by the EDA tools.

#### 5.3.2.3 Dependable DPR with Cyclic Redundancy Check ($D^2PR$-$CRC$)

The first approach aims at detecting partial bitstream data errors by performing periodic on-line CRC checks. As shown in figure 5.15, at design-time the partial bitstream is parsed and processed by a software routine in order to compute and embed CRC signatures, similarly to what done in Section 5.2.

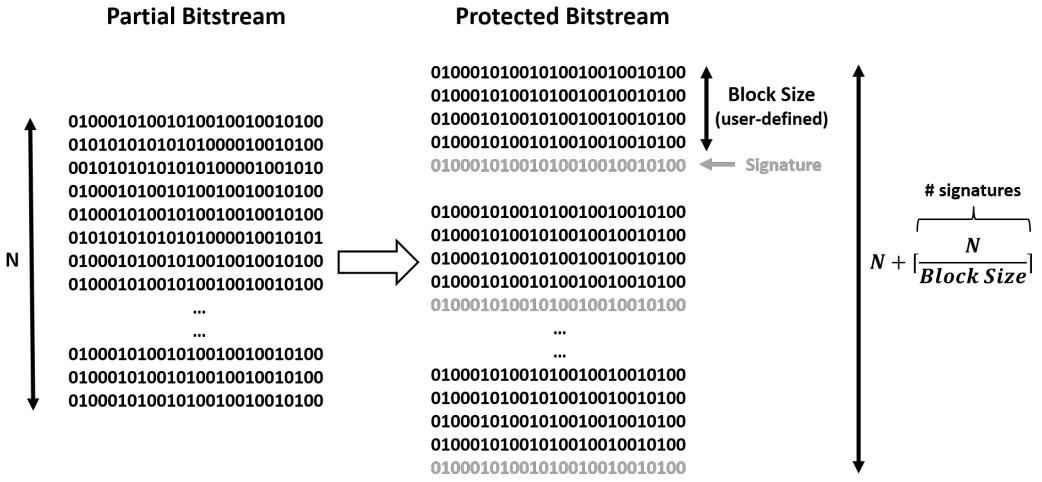



Figure 5.15: Protected bitstream generation.

In particular, the partial bitstream is split in blocks of 32-bit words. A signature is then computed for each data block and embedded in the *Protected Bitstream*.

Figure 5.16 shows the architecture of the DPR controller when configured in $D^2PR$-$CRC$ mode. With respect to the *Asynchronous* mode, the *FSM READ* is assisted by an on-line *CRC Generator* and a *CRC checker*. Basically, whenever a *Protected Bitstream* word is read from the memory through the DMA, it is directly written in the FIFO buffer. In addition, it is sent to the *CRC gener-*

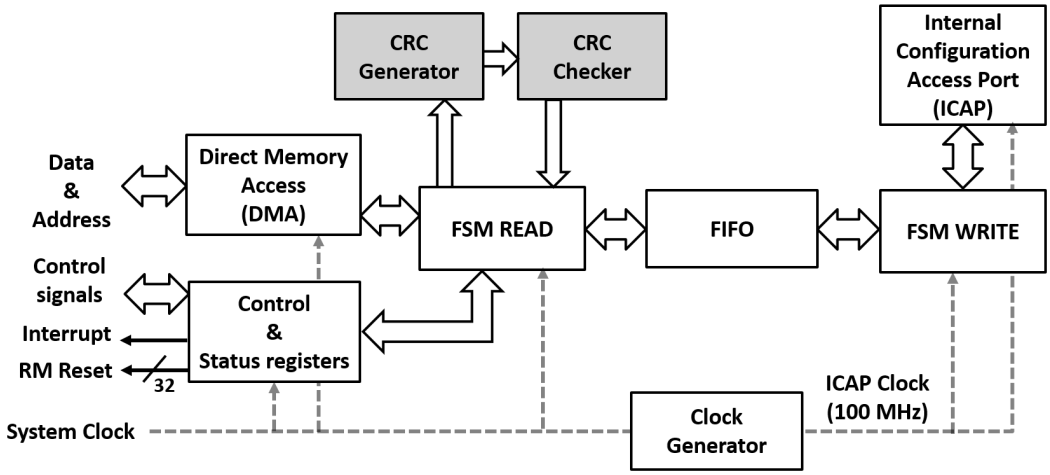



Figure 5.16: DPR controller architecture for D$^2$PR-CRC mode.

*ator*, implemented as a parallel 32-bit Linear Feedback Shift Register (LFSR) [13], that computes at run-time a 32-bit signature in a single clock cycle.

Whenever a data block is completely received, the associated pre-computed signature is read trough the DMA and compared with the one extracted at run-time by the *CRC generator*. If no errors are detected, the currently received data block is validated and the *FSM WRITE* can start reading it from the *FIFO* in order to deliver bitstream data to the ICAP. While the buffer is being emptied, *FSM READ* instructs the DMA engine to retrieve the following data block.

Obviously, the FIFO buffer must be sized in order to store at least one full data block. The data block size can be configured by the user at design-time. This parameter has a direct impact on (i) bitstream storage memory requirements, (ii) actual reconfiguration throughput, and (iii) error detection capabilities.

Equation 5.7 shows the relationship between the total Protected Bitstream Size ($PBS$) and the user-defined data block size ($Blk$).

$$PBS = BS + 32 \cdot \lceil \frac{BS}{BlkS} \rceil \tag{5.7}$$

The second term of Equation 5.7 represents the additional contribution given by the 32-bit signatures interleaved in the partial bitstream.

The maximum effective reconfiguration throughput that can be achieved by the proposed architecture can be estimated by the following equation:

$$Max\_TP = \frac{BS}{max\{T_{sys}, T_{ICAP}\} \cdot \frac{PBS}{32} + T_{ICAP} \cdot \frac{BlkS}{32}} \tag{5.8}$$

In Equation 5.8, the first term of the denominator represents the overall time needed to read the *protected bitstream* through the DMA, while the second term consider the additional latency,

introduced by the adopted buffering approach, for delivering the last data block to the ICAP after its validation.

The actual reconfiguration throughput depends also on bus and memory bandwidths, as reported by Equation 5.5.

The 32-bit CRC error detection approach provides detection of all burst errors, up to 32 bits in a single data block, and a tunable coverage on random errors depending on the chosen CRC polynomial and data block size [114] (see Section 5.3.3).

The same protection strategy discussed for the *Asynchronous* mode can be employed to protect the proposed architecture against faults affecting the device. In particular, since CRC checks are performed before writing data in the FIFO, the additional parity checks (see Section 5.3.2.1) are needed to detect errors affecting data stored in the Block-RAMs used to implement the buffer.

Finally, it is worth noting that, with respect to the *PerFrameCRC* functionality offered *only* by *7Series* FPGAs [238], the proposed approach represents a more flexible solution, allowing designer to tune memory requirements, error detection capabilities and timing overhead depending on the constraints imposed by the target application.

#### 5.3.2.4 Dependable DPR ($D^2PR$) with Error Correcting Code ($D^2PR$-*EDAC*)

In $D^2$*PR-EDAC* mode, the proposed controller provides partial bitstream error detection and correction capabilities through an Error Correcting Code (ECC). In particular, a Single Error Correction Double Error Detection (SECDED) Error Correcting Code (ECC) has been chosen due to its representative target error model and its limited code and hardware overheads [82].

Partial bitstreams must be first processed at design-time by a software routine in order to produce a *Protected bitstream* composed of SECDED encoded words packets, that will be subsequently decoded by the DPR controller at run-time.

As shown in Figure 5.17, each *Protected bitstream* packet is composed of 5 32-bit words, where Encoded$i(x:y)$ represents the $(x,y)$ bit range of the encoded word associated to the 32 bit bitstream Word$i$. It is worth to remember that SECDED encoding process results in 7 bits overhead on a 32 bits input word [82].

The architecture of the DPR controller configured in $D^2$*PR-EDAC* mode is depicted in Figure 5.18.

Taking as reference the architecture presented for the *Asynchronous* mode (see Section 5.3.2.1), the *FSM WRITE* module is assisted by an on-line SECDED decoder, that checks and decode the received words and, if a single error is found, correct them.

*Protected bitstream* data packets are continuously read through the DMA and 32-bit encoded data words are written in the FIFO, only needed if system clock differs from the chosen ICAP clock. *FSM WRITE* continuously reads out data from the buffer until an entire 5-words packet is reconstructed. Whenever the fifth word of a packet is read out, the decoding phase starts. It

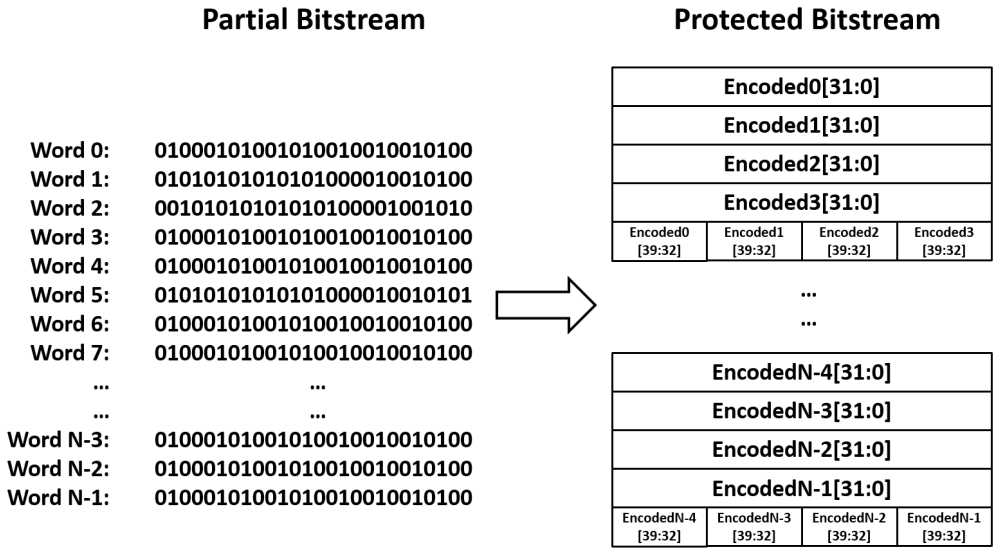



Figure 5.17: Protected bitstream generation.

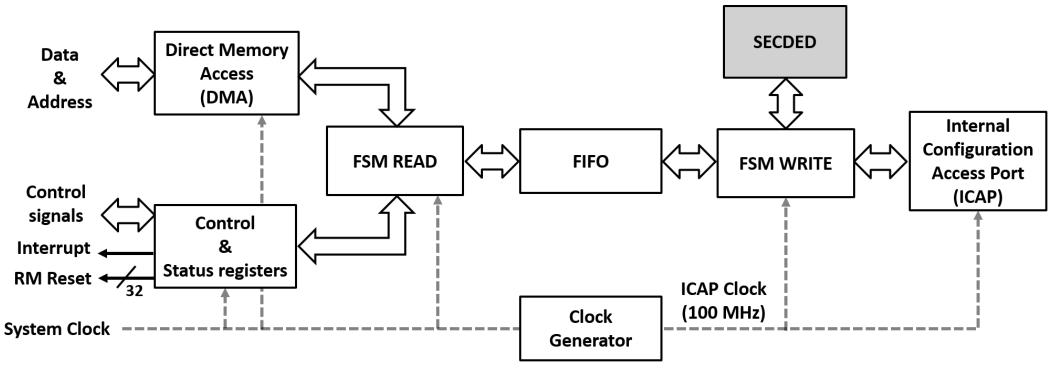



Figure 5.18: DPR controller architecture for D$^2$PR-EDAC mode.

consists lasts four clock cycles, needed to decode four bitstream data words. At each clock cycle a data word is decoded and delivered to the ICAP. If present, any single error is corrected before writing the data word in the configuration port. On the other hand, the reconfiguration process is terminated if double errors are found during the decoding process. At the end of this phase, *FSM WRITE* restarts the process by reading the following 5-words packet from the FIFO.

Similarly to the DPR controller operating in ($D^2$*PR-CRC*) mode, the SECDED *Protected Bitstream Size* and the maximum throughput achievable by the proposed controller can be estimated using Equations 5.5-5.8. However, in this operating mode, data block size ($BlkS$) is fixed and equal to 128 bits.

Integrating error correction in the DPR controller allows to avoid interruptions during re-

configuration processes due to bitstream data errors, therefore preserving overall system performances.

A protection strategy similar to the one discussed for the *Asynchronous* mode can be adopted to protect the proposed architecture against faults affecting the device. In particular, since bitstream data words are checked just before being delivered to the ICAP, the additional parity encoder and decoder embedded in *FSM READ* and *FSM WRITE* (see Section 5.3.2.1) are not needed. In fact, under the assumption of single or double occurrences, errors affecting data stored in the Block-RAMs used to implement the FIFO buffer are also detected by the SECDED logic.

### 5.3.3 Experimental Results

To demonstrate the flexibility of the proposed controller and evaluate its hardware overheads and reconfiguration throughput, two different test cases have been implemented. In the first scenario, the proposed DPR controller has been instantiated in a LEON3 processor-based system [85] implemented on a *Virtex4-VLX100* FPGA. As shown in Figure 5.19, the controller is connected to the on-chip AMBA AHB bus, acting as a master peripheral.

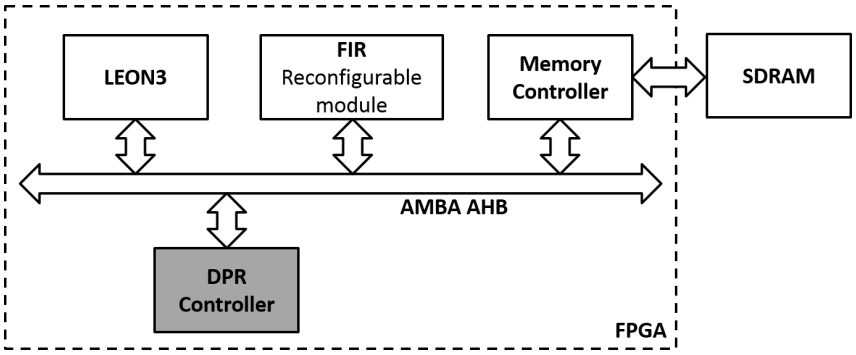



Figure 5.19: Proposed controller instantiated in a LEON3-based system.

A small wrapper is needed to adapt the interface of the *DMA* module to the signals required by the AMBA AHB protocol. On the other hand, the registers included in the *Control and Status registers* block have been mapped in the bus memory space, allowing the LEON3 processor to read and write them through simple software drivers. The DPR controller is used to reconfigure a FIR filter peripheral physically mapped to an FPGA location requiring 28.7 KBytes of bitstream data in order to be reconfigured. Bitstream data are stored in an external SDRAM memory, accessible by the *DMA* engine through a memory controller mapped on the bus. Reconfigurations are triggered and monitored by the processor.

Reconfiguration time and throughput have been measured using an hardware timer embedded in the DPR controller. Table 5.2 reports the obtained results, in terms of hardware resources and effective reconfiguration throughput, when implementing the proposed controller in the

four different configurations presented in Section 5.3.2. System and ICAP frequencies are set to 60 MHz and 100 MHz, respectively. In Table 5.2, percentages denote resources consumption

Table 5.2: Hardware resources and throughput for the proposed controller operating in a LEON3-based system implemented on a *Virtex4-VLX100* FPGA.

| **Configuration** | **LUTs** | **FFs** | **BRAMs** | **Throughput [MBytes/s]** |
|---|---|---|---|---|
| *Synchronous* | 307 **(0.3%)** | 118 **(0.1%)** | - | 235 **(240)** |
| *Asynchronous* | 471 **(0.5%)** | 235 **(0.2%)** | 1 **(0.4%)** | 235 **(240)** |
| *$D^2$ PR-CRC* | 811 **(0.8%)** | 318 **(0.3%)** | 1 **(0.4%)** | 229.7 **(235.8)** |
| *$D^2$ PR-EDAC* | 815 **(0.8%)** | 352 **(0.5%)** | 1 **(0.4%)** | 186 **(192)** |

with respect to the ones available in the target device. In all configurations, the proposed controller provides very low hardware overhead. This is an important aspect when adopting DPR for reducing overall system resource requirements by time-multiplexing hardware modules. In fact, FPGA logic used by the DPR controller must reside in the static portion of the design, thus representing a hardware overhead of the system. Obviously, a digital clock manager is also required to generate the 100 MHz ICAP clock needed in *Asynchronous*, *$D^2$ PR-CRC* and *$D^2$ PR-EDAC* operating modes.

The measured throughput values are compliant with the ones estimated (reported between parentheses in Table 5.2) exploiting Equations 5.5 and 5.8. AMBA AHB bus and memory controller latencies cause low overhead during data transfers and the main parameter that limits the reconfiguration throughput is represented by the system clock frequency (i.e., 60 MHz).

When configured in *$D^2$ PR-CRC* mode, the CRC polynomial implemented by the *CRC generator* is $x^{32}+x^{18}+x^{14}+x^3+1$. This particular polynomial provides detection of all burst errors up to 32 bits in a data block and detection of all 5 random errors for data blocks smaller than 31 Kib [114]. For the chosen implementation, the data block size has been set to 108 32-bits words (i.e., 3.37Kib), allowing the usage of a single Block-RAM to implement the *FIFO* buffer.

In general, when operating in *$D^2$ PR-CRC* mode, the actual effective throughput depends on the chosen data block size, as reported by Equation 5.8. Figure 5.20 reports the trend of the reconfiguration throughput with respect to the block size, considering $BS = 28.7KiBytes$, $T_{sys} = 16.66ns$ and $T_{ICAP} = 10ns$.

As shown in Figure 5.20, the reconfiguration throughput can be maximized if properly selecting the data block size (in the considered test case the optimal block size is equal to 108 partial bitstream words). By looking at Equation 5.8, it can be also demonstrated that the block size that maximizes the throughput depends on the size of the original unprotected bitstream (i.e.

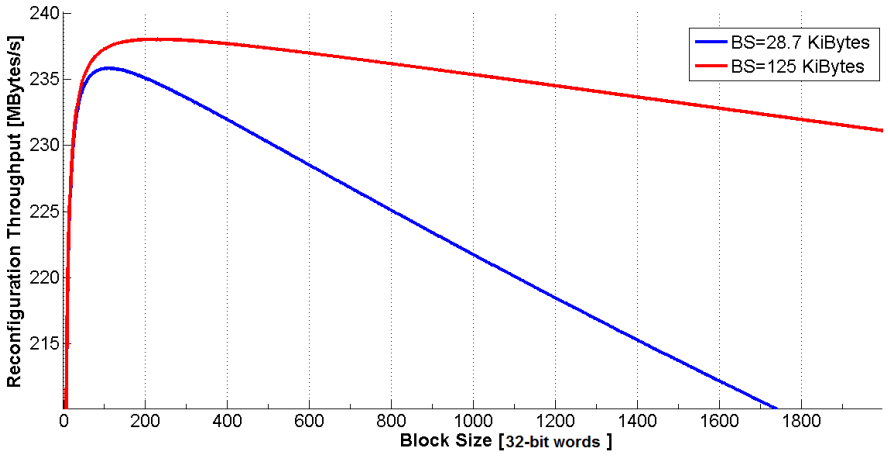



Figure 5.20: Reconfiguration throughput w.r.t. data block size for the proposed DPR controller configured in $D^2$*PR-CRC* mode.

$BS$). This is also shown in Figure 5.20, where the red line represents the trend of the reconfiguration throughput w.r.t. the chosen block size when $BS = 125KiBytes$. Results shown in Table 5.2 highlight that it is possible to monitor bitstream data errors incurring in limited reconfiguraton timing overheads if the optimal bitstream data block size is selected at design-time when the proposed DPR controller is configured in $D^2$*PR-CRC* mode. The reconfiguration time overhead in $D^2$*PR-EDAC* mode is caused by the additional bits needed after the bitstream encoding process. However, as discussed in Section 5.3.2.2, the DPR controller configured in $D^2$*PR-CRC* mode provides the same performances of $D^2$*PR-EDAC* mode if data block size is set to 128 bits (i.e., 4 bitstream words).

The second scenario is represented by a system in which the DPR controller is directly interfaced to an on-chip memory, implemented using Block-RAMs, storing the partial bitstream. In this case the user logic must trigger and monitor the reconfiguration process by reading/writing the values in the *Control and Status registers* block.

The system has been implemented on an *Artix7-xc7a100t* FPGA. Table 5.3 reports the hardware resources and throughput of the proposed controller implemented in its four possible configurations. In this case, System frequency is set to 200 MHz, ICAP frequency is fixed to 100 MHz, and the size of the bitstream needed to change the functionality of the selected reconfigurable FPGA portion is equal to 119.4 KiBytes. In $D^2$*PR-CRC* mode $BlkS = 5.5Kib$ to maximize the reconfiguration throughput.

The measured values match the ones estimated using Equations 5.5 and 5.8, since the DPR controller is directly interfaced to a fast on-chip memory, and the throughput bottleneck is represented by the maximum ICAP clock frequency (i.e., 100 MHz).

Table 5.3: Hardware resources and throughput for the proposed controller implemented on an *Artix7-xc7a100t* FPGA.

| Configuration | LUTs | FFs | BRAMs | Throughput [MB/s] |
|---|---|---|---|---|
| *Synchronous* | 249 **(0.4%)** | 112 **(0.1%)** | - | 400 |
| *Asynchronous* | 443 **(0.7%)** | 226 **(0.2%)** | 1 **(0.7%)** | 400 |
| *$D^2$ PR-CRC* | 588 **(0.9%)** | 278 **(0.2%)** | 1 **(0.7%)** | 395.4 |
| *$D^2$ PR-EDAC* | 589 **(0.9%)** | 310 **(0.2%)** | 1 **(0.7%)** | 319.9 |

The proposed DPR controller has been also implemented on a *Virtex6-vlx240t* FPGA, in order to be compared with other portable state-of-the-art and vendor solutions. Results are reported in Table 5.4, adopting the same input parameters of the previous scenario.

With respect to state-of-the-art and vendor solutions, the proposed controller provides similar hardware overheads while being able to monitor and eventually correct bitstream data errors. At the same time it is able to sustain high reconfiguration throughputs. It is worth noting that both *XPS_HWICAP* and *AXI_HWICAP* can be used only in conjunction with processor-based systems, since they show an OPB or AXI bus slave interface. The processor or an external master in the bus is directly in charge of managing bitstream data trasfer, thus leading to increased timing overheads. The proposed controller improves also vendor solutions by providing portability on different FPGA device families and high configurability, while being able to achieve the maximum possible reconfiguration throughput.

Its robustness with respect to faults affecting the device can be improved by applying dual or triple modular redundancy or one the approaches proposed in [71]. When targeting standard

Table 5.4: Comparison of the proposed DPR controller with state of the art and vendor solutions. The target device is a *Virtex6-vlx240t* FPGA.

| Implementation | LUTs | FFs | BRAMs | Throughput [MB/s] |
|---|---|---|---|---|
| [211] | 586 | 672 | 8 | 399.8 |
| [199] | 673 | 254 | 5 | 253 |
| *XPS-HWICAP*[227] | 799 | 746 | 1 | 8.5 |
| *AXI-HWICAP*[235] | 502 | 477 | 1 | 9.1 |
| *Synchronous* | 249 | 112 | - | 400 |
| *Asynchronous* | 443 | 226 | 1 | 400 |
| *$D^2$ PR-CRC* | 588 | 278 | 1 | 395.4 |
| *$D^2$ PR-EDAC* | 592 | 310 | 1 | 319.9 |

replication approaches, the method proposed in Section 5.3.2 can be applied to avoid any memory resources overhead. In this case, additional hardware overheads due to the introduction of parity encoders and decoders must be taken into account. Moreover, with respect to a full replication approach (e.g., TMR replicating the entire DPR controller, including memory buffer), an additional intermediate voter is needed at the input of the FIFO (as shown in Figure 5.14). Due to the very low complexity of the adopted parity scheme, these additional components lead to a negligible hardware overhead with respect to the resources needed to accommodate all the DPR controller replicas. Moreover, as discussed in Section 5.3.2.2, when the proposed controller is configured in $D^2$*PR-SECDED* mode, parity logic is not needed since bitstream data are checked by the SECDED logic just before being delivered to the ICAP. On the other hand, the proposed approach does not cause an increment of required FIFO resources, consuming the 50% or 33% of memory resources if compared to standard dual or triple modular redundancy schemes, respectively.

CHAPTER 6

# EVALUATING SYSTEM'S ROBUSTNESS THROUGH ERROR INJECTION

How mentioned in Chapter 3, evaluating system performances and robustness is mandatory when targeting safety and or mission critical systems. When employing FPGAs in such systems particular attention must be taken. Radiation induced SEUs can affect both the memory elements embedded in the design, and the configuration memory that stores the related FPGA configuration. In the former case, SEUs may alter the content of the FPGA internal memory resources employed in the design (e.g., flip-flops, distributed RAMs, or Block-RAMs), modifying processed data and/or the application control flow, thus leading to problem similar to the ones occurring in ASICs. In such cases fault injection techniques similar to the ones used when dealing with ASICs can be employed to assess the reliability of the design (e.g., scan-chains [42], or saboteurs [138]). Instead, the latter case is much more critical, since a bit-flip in an FPGA configuration memory cell may permanently alter the functionality of the implemented circuit. In particular, it can cause changes in the configuration of Look-Up Tables (LUTs), Configurable Logic Blocks (CLBs), internal hard macros (e.g., Digital Signal Processors or Block-RAMs), or routing matrices, leading to completely different circuits from the initially configured ones [37]. Furthermore, the configuration memory is very large compared to all the other elements in the device. Therefore, the probability that SEUs affect the FPGA configuration memory is high, making it a major concern when designing high reliable FPGA-based systems. Nevertheless, it has been demonstrated that, in most cases, SEUs affecting the configuration memory do not influence the design functionality [126, 226, 246]. Consequently, fault injection techniques are needed to deeply analyze the effect that SEUs in the configuration memory have on the implemented design functionality. Fault injection can help designers to discover the weaknesses of the circuit and to take the proper countermeasures by applying the most suitable fault detection and/or fault tolerance techniques to selectively and efficiently harden the design.

To accurately assess the reliability of FPGA-based systems, designers must rely on very expensive neutrons or heavy ions beam radiation tests on actual circuit prototypes [81, 192]. Preliminary, even less accurate, fault injection experiments in the early design stages can potentially reduce the number of design iterations, speeding up the entire design process of complex Systems on a Programmable Chip (SoPC).

In literature, several solutions tackling this problem have been proposed and developed, spanning from simulation-based methods [37] to hardware approaches [18, 105, 166]. The latter approaches guarantee the maximum fault injection speed when the circuit under test and the fault injection infrastructure are implemented in the same FPGA. To achieve high fault injection rates they usually exploit DPR feature of modern SRAM-based FPGA devices (i.e., the ability to dynamically change selected portions of a circuit, while the rest of the design is left unchanged and fully functional [238]). Nonetheless, faults injected using this approach can affect the operations of the fault injection infrastructure itself, or they can cause faults accumulation effects. In fact, in some unpredictable conditions, the reference gold circuit state, i.e., the circuit state in which the next fault should be injected, cannot be properly restored [116, 166]. This can lead to the stall of the fault injection process, or to unknown erroneous faults classification results.

This chapter illustrates a Dynamic Partial Reconfiguration-based fault injection infrastructure for SEUs emulation in the configuration memory of Xilinx SRAM-based FPGAss. The proposed infrastructure is integrated in the same device as the system under test, and exploits the Xilinx *Essential Bits* technology [237] to:

1. extremely speed-up the fault injection process, as demonstrated by experiments carried out on designs of different complexity;

2. ensure the correct behavior of the fault injection infrastructure itself and avoid undesired faults accumulation effects during the whole fault injection process, as it occurs on other single-FPGA fault injection platforms [18, 116, 166].

## 6.1 Related Works

The approaches for performing fault injection in the configuration memory of FPGA devices can be grouped into three main categories (Fig. 6.1).

The first group includes techniques that exploit the availability of radiation test environments, in which neutrons or heavy ions beams are used to emulate the radiation conditions in which the device will operate [81, 192]. Since radiation sessions are very expensive, their use is limited to the validation of a system in the final design stages. Moreover, these experiments provide low controllability, since the memory cell to be flipped cannot be chosen, and the injection rate is lower with respect to other fault injection techniques [81].

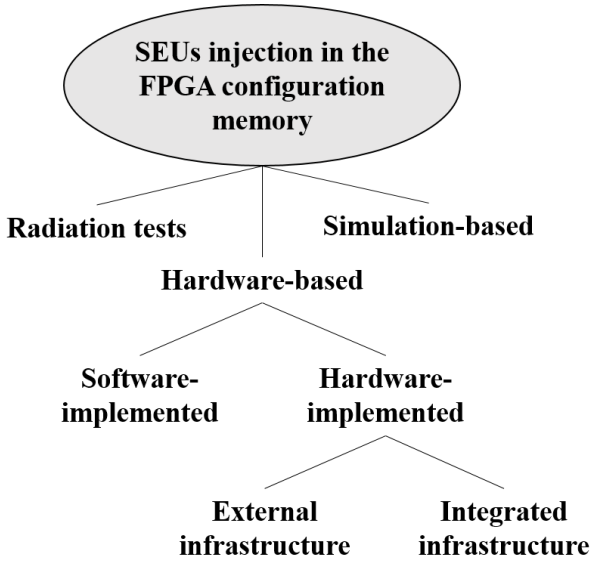



Figure 6.1: FPGA configuration memory SEUs fault injection approaches classification

The second group includes simulation-based approaches, that do not require the physical FPGA device. Simulations are performed relying on models that link each bit of the FPGA configuration memory to the associated functionality at logic level. However, since this information is not disclosed by FPGA vendors, models must be indirectly retrieved by reverse-engineering experiments [37]. Furthermore, simulation-based fault injection techniques incur in huge simulation time to obtain statistically valid results [21, 108].

The third, and biggest, category includes those methods that use extra hardware, or software, to inject SEUs in the configuration memory of the FPGA. Basically, the device under test is configured with a corrupted *bitstream* configuration file, in which a bit-flip is randomly inserted to emulate an SEU. To dramatically reduce the injection time, Dynamic Partial Reconfiguration is used to reconfigure the minimum possible portion of the configuration memory, called *frame*, instead of the entire device. This category includes Software-Implemented Fault Injection (SWIFI) methods, that exploit a host PC, running a program managing the reconfiguration of the device under test and the entire fault injection process [22, 89, 106, 112, 194], and Hardware-Implemented Fault Injection (HWIFI) methods, that instead use extra-hardware (external or integrated in the same FPGA of the system under test) to further accelerate the fault injection process [3, 18, 105, 116, 125, 151, 166].

Due to the increased density and capacity, in terms of logic cells, of modern FPGAs, HWIFI methods integrated in the same FPGA of the system under test are nowadays the preferred solutions. In fact, they do not require external hardware and provide the maximum possible fault injection speed. They enable to gain several orders of magnitude in the injection speed, with

respect to SWIFI and external HWIFI techniques.

Basically, for each fault injection run, the infrastructure reads a *frame* of the configuration memory, composed of several 32-bit words [233]. Then it modifies the *frame* according to the chosen fault model, and reconfigure the configuration memory with the faulty *frame*. The internal fault injection infrastructure also provides the input vectors and reads the outputs of the system under test to verify their correctness. Finally, the faulty *frame* is restored and another bit in the same or in another frame is flipped [18, 166]. To accomplish the reconfiguration task, the ICAP is employed [238].

Nonetheless, to properly operate, these approaches require an in-depth knowledge of the structure and the frames addressing order of the considered FPGA. In fact, the designer must know the addresses of the configuration frames associated with the FPGA resources implementing the system under test. This information is not explicitly provided by vendors and depends on the specific FPGA model. Moreover, in some unpredictable conditions, since a bit-flip in a *frame* can affect also the information in the other frames, the fault-free configuration cannot be restored after fault injection [166]. In this case, a reconfiguration of the entire device is therefore needed to avoid fault accumulation effects [116]. Furthermore, extra care must be taken since faults injected using the integrated HWIFI approaches can affect the operations of the fault injection infrastructure itself. This unintended effect can lead to the stall of the fault injection process, or to unknown erroneous faults classification results [116].

The proposed integrated HWIFI infrastructure overcomes these limitations by exploiting the *Xilinx Essential Bits* technology [237], and Dynamic Partial Reconfiguration, to ensure the correct behavior of the fault injection infrastructure itself, and to dramatically speed-up the fault injection process reducing the set of fault locations. Moreover, it avoids any undesired fault accumulation effect during the whole fault injection process, still providing high fault injection rates.

## 6.2 Proposed Methodology and Infrastructure

The architecture of the proposed fault injection infrastructure is depicted in Figure 6.2.

It is mainly composed of: (i) the *Fault Generator*, (ii) the *System Clock Controller*, (iii) the *System Input Controller*, (iv) the *System Output Collector*, and (v) the *Fault Classifier*. Moreover, a *Main Control Unit* manages, synchronizes and coordinates the activities of all the aforementioned modules.

The infrastructure takes in input the bitstream configuration file of the Circuit Under Test (CUT), the *Input Vectors* needed to exercise the CUT, and the *Golden Outputs* of the fault-free CUT run. According to the SEU fault model, a single bit-flip is introduced in the configuration memory for each execution run. At the end of the fault injection process, the infrastructure pro-

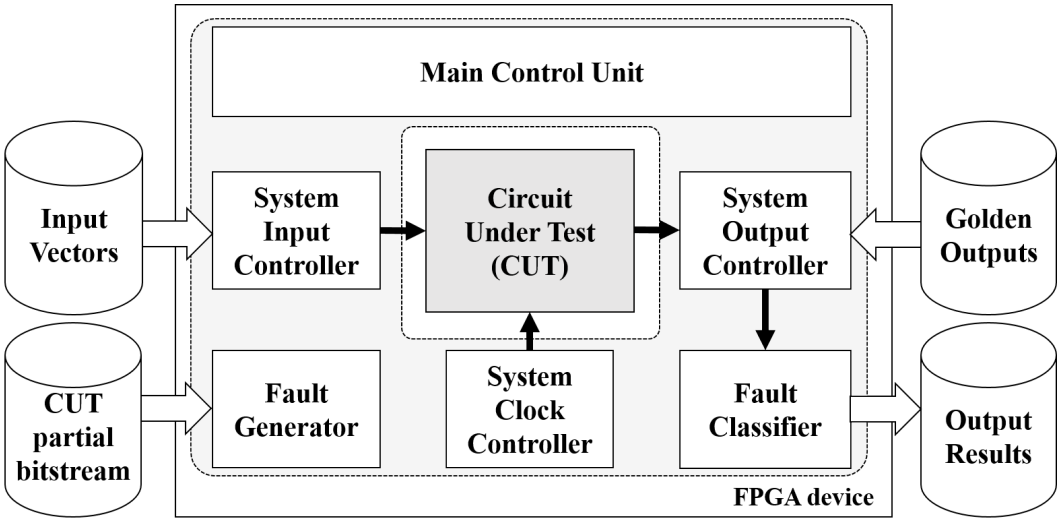



Figure 6.2: Proposed fault injection infrastructure architecture

vides in output the results in terms of percentages of faults that caused an observable functional error in the CUT.

The next subsections details the operation of each module composing the proposed infrastructure.

### 6.2.1 Fault Generator

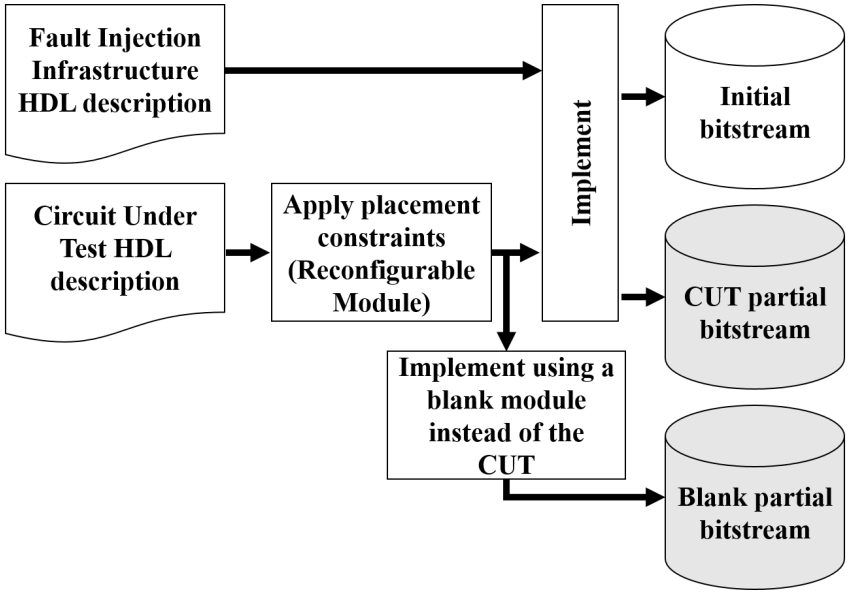
According to the *FARM* model [11], when designing a fault injection environment, one must take into account the adopted *F*ault Models, the *A*ctivation patterns used to stimulate the system under test, the *R*eadouts values collected during the experiment, and the extracted *M*easures.

As aforementioned, the fault model adopted by the proposed infrastructure is the SEU in the configuration memory of the FPGA device. The choice of which configuration bit must be flipped is randomly made on-line, during each fault injection run, choosing from a set of possible locations, generated off-line.

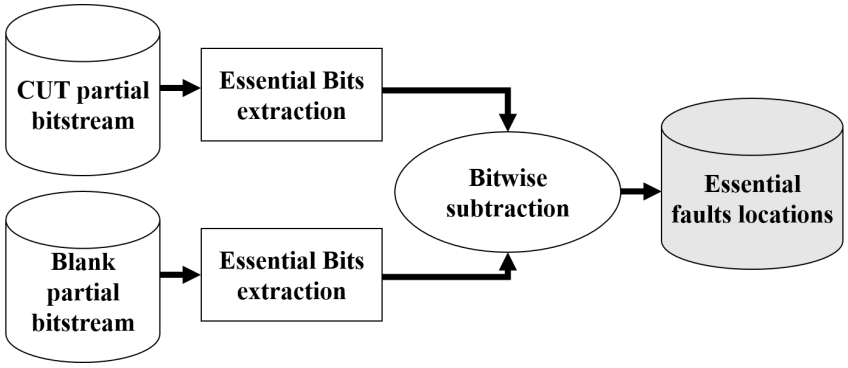
The set of possible locations in which the SEUs will be injected is generated in two steps (Figure 6.3).

To ensure the correct operations of the fault injection infrastructure during the whole fault injection process, the basic idea of the proposed approach is to implement the CUT in a reconfigurable region of the FPGA, and to extract, from the bitstream, only the locations of the bits associated to the CUT. Differently from the CUT, the fault injection infrastructure is instead implemented in a non-reconfigurable, or *static*, portion of the device.

During the first phase (Figure 6.3(a)), the HDL description of both the fault injection infrastructure and the CUT are merged and implemented to obtain the *Initial bitstream* configuration




(a) Step 1: CUT and blank partial bitstream generation.




(b) Step 2: essential faults location generation.

Figure 6.3: Fault locations generation flow

file, used to configure at startup the entire FPGA.

In order to implement the CUT in a reconfigurable partition, before the actual implementation, some placement constraints must be provided (i.e., the designer allocates a defined area of the FPGA to the CUT). In this case the CUT is called *Reconfigurable Module*. After that, the implementation process generates also the CUT *partial bitstream* file, that contains the information needed to configure the *Reconfigurable Module*.

The same process is repeated to implement an empty, or *blank*, reconfigurable module instead of the CUT, in the same previously defined reconfigurable area. The *Blank partial bitstream*

file associated with this module contains only the information regarding the input/output interfaces of the CUT (called *partition pins*), and the routing associated to the static part of the design that passes through the reconfigurable partition. As stated in [238], only routing, and not logic, associated with the static part of the design can use hardware resources contained in a reconfigurable partition.

Consequently, since this static routing information cannot be prevented to pass through reconfigurable partitions, an SEU injected in the configuration bits associated to this area can cause errors or failures also in the fault injection infrastructure itself (i.e., the static part of the design) [238]. This can make the fault injection infrastructure unusable, or behaving incorrectly.

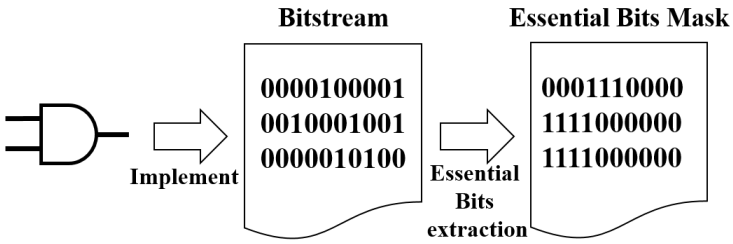
To ensure that no faults are injected in a configuration memory cell associated with the fault injection infrastructure, a further step is needed. This second step (Figure 6.3(b)) represents the main difference with respect to existing state-of-the-art approaches that do not tackle the aforementioned static routing problem.

Starting from the partial bitstreams associated to the CUT and to the blank reconfigurable modules, the *Xilinx Essential Bits* technology is used to restrict the set of potential fault injection locations[237]. This technology offers a functionality, embedded in the Xilinx *BitGen* tool [234], that allows to identify and to extract, from the bitstream files, the configuration bits that are essential to the design functionality (i.e., the so called *essential bits*). In fact, only a small fraction of the bits are essential to the proper operation of any specific design loaded into the FPGA device [237]. As shown in Figure 6.4(a), the *Essential Bits* functionality provides a mask in which if a bit is set, the associated bit in the bitstream file is essential, and thus, if flipped, it modifies the circuit functionality (Figure 6.4(b)).

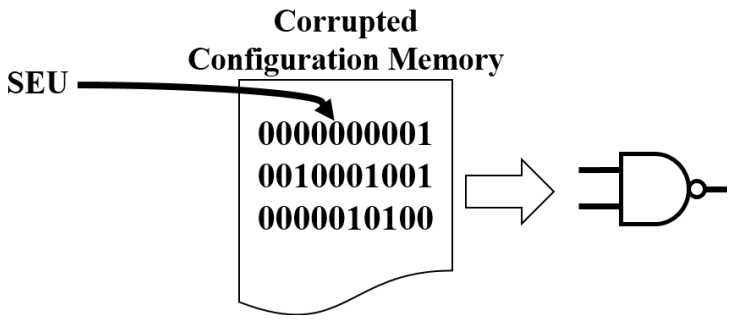
Since the objective of a fault injection infrastructure is to flip those bits associated with the CUT, resorting to the essential bits mask of the blank partial bitstream it is possible to localize the locations of the bits associated with the static routing passing through the reconfigurable module. The mask containing the position of the essential bits of the CUT (*Essential faults locations* in Figure 6.3(b)) is obtained by bit-wise subtracting the masks associated to both the CUT and the Blank partial bitstreams. It is worth to remember here that, as aforementioned, some bits in the Blank bitstream carry information about the I/O interfaces of the CUT. By applying this method, these bits will be not flipped during the fault injection process, thus their contribution represents an error on the final computed fault injection metrics. However, as will be demonstrated in Section 6.3, this contribution is very small, thus can be assumed negligible if the fault injection experiments are made in the early design stages, where highly accurate results are not required.

This means that the fault injection infrastructure will always inject an SEU in a position associated with a bit that has no impact on the infrastructure itself, thus ensuring correct operations during the whole injection process.

The second main advantage of using the *Essential Bits* technology for fault injection is that




(a) FPGA design, associated bitstream, and essential bits mask.




(b) Change of functionality due to an SEU in the configuration memory cell storing an essential bit of the implemented design.

Figure 6.4: *Essential bits* meaning

the injection time is dramatically reduced, with respect to inject faults in every possible memory location. In fact, in general, the essential bits of an implemented module are lower than 20% of the total configuration bits [237]. Thus, it is useless to inject SEUs in the remaining 80% of the bits composing the bitstream, since it is known a-priori that those faults will not cause any observable functional error. Obviously, these not injected faults are counted as if they have been injected to compute the final output statistics results.

The *Essential faults location* mask is used at run-time by the *Fault Generator* to choose a possible fault injection location. Using a LFSR, a pseudo-random injection time and location (among the possible given by the *Essential faults location* mask) are generated. When the execution time reaches the selected injection time, the *Fault Generator* reads the fault-free CUT partial bitstream (Figure 6.2) and flips a bit according to the selected position. Afterwards, the execution can be resumed. At the end of the execution, the CUT reconfigurable module is restored with the fault-free partial bitstream. This last reconfiguration overwrites all the configuration *frames* associated with the CUT, avoiding any fault accumulation effect.

It is worth noting that the proposed fault generation method can be easily adapted to emulate also MBUs. This fault model is expected to become a more realistic model, with respect to SEUs,

for modern and future technology nodes [88].

### 6.2.2 System Input Controller

Referring to the *FARM* model [11], the Activation patterns, or *Input Vectors*, are provided to the CUT through the *System Input Controller*. The *Input Vectors* can be stored either internally to the FPGA device or in an external memory, depending on their size and on the CUT nature (e.g., processor, datapath, control unit). They can be generated off-line through simulations or other techniques. Thus, the *System Input Controller* consists of a control unit, acting as a memory controller, that reads the vectors from an internal or external memory, and feeds the inputs of the CUT. When the CUT needs pseudo-random input vectors, the *System Input Controller* simply consists of a LFSR [21].

### 6.2.3 System Clock Controller

The *System Clock Controller* is mainly composed of one or more FPGA Digital Clock Managers (DCMs) [240], able to synthesize and manage the clocks needed by the CUT.

It works in conjunction with the *System Input Controller* to synchronize the CUT clock to the *Input Vectors*, and to stop it during the reconfiguration process. In particular, at a random time, during the execution of a run, the CUT clock is stopped and the configuration memory is reconfigured using the faulty CUT partial bitstream, generated as explained in Subsection 6.2.1. Afterwards, the CUT clock is re-activated, resuming the execution of the actual run until the fault is detected, or until the end of the run, if the injected fault does not generate an observable error.

### 6.2.4 System Output Collector and Fault Classifier

Referring to the *FARM* model [11], the Readouts and the Measures are performed through the *System Output Collector* and *Fault Classifier* modules.

The *System Output Collector* monitors the outputs of the CUT after each fault injection. Depending on the number of outputs and number of responses to observe, the designer can choose the output comparison technique that best fit the considered CUT and test case (e.g, outputs compression and signatures comparison, clock-by-clock comparison [21], etc).

If a difference is encountered during the outputs comparison process, the fault in the configuration memory of the FPGA device is targeted as *critical*. In the opposite case, the fault is considered *non-critical* since, even if at the end of the run it can be still present in the configuration memory, it does not generate erroneous results or system failures. If the CUT is equipped with a fault detection mechanism, the error detection signal can be used by the *Fault Classifier* to classify the faults detected by the CUT detection mechanisms, labeling them as *hardware detected*.

This functionality can be very useful when, in the early design stages, the designer is interested in evaluating different fault detection techniques.

Eventually, the *Fault Classifier* can record both the position of the flipped bit in the CUT partial bitstream and the injection time. This enables a further post-processing, since this information can be merged with the FPGA *Logic Allocation File*. This file can be generated by the *Xilinx BitGen* tool, and it provides information (not including routing) related to which design module is associated with a set of bits in the bitstream [234].

## 6.3 Experimental Results

This section presents the experimental results gathered by implementing the proposed fault injection infrastructure on a *Xilinx ML605* evaluation board, equipped with a *Virtex-6 LX240T* FPGA [228].

Three case studies have been considered:

- *LEON3* processor [84], running several applications extracted from the *MiBench* benchmark suite [96]. The applications have been selected in order to stimulate different units of the processor. In particular, the selected applications are: the Susan Edge detector, that extensively exploits the arithmetic operations of the Integer Unit and the Load/Store Unit, the CRC32, that makes large use of the boolean operations of the Integer Unit, and the Inverse Fourier Transform (IFFT), that mainly uses the floating-point Unit.

- two-dimensional convolution datapath, as the one reported in [36], composed of 49 8x8 multipliers and a balanced adder tree including 48 adders. This architecture is often employed when dealing with two-dimensional images filtering;

- the same aforementioned two-dimensional convolution datapath, with Triple Modular Redundancy (TMR) error correction and detection mechanism, applied as shown in Fig. 6.5 (the majority voter sets the *Error detected* signal when it recognizes a mismatch between the three module outputs).

Tables 6.1, 6.2, and 6.3 report the hardware resources, in terms of Look-Up Tables (LUTs), Flip-Flops (FFs), Block-RAMs (BRAMs), and Digital Clock Managers (DCMs), needed to implement the CUTs and the associated infrastructures in the three considered test cases, respectively.

For the *LEON3* case study, the processor runs at 80 MHz, reading the application and the data from an external memory through the *System Input Controller*. The *System Output Controller*, at the end of the execution, reads from the external memory the results produced by the processor, and compares them with the golden ones stored in an internal Block-RAM. The comparison results are sent to the *Fault Classifier* which computes the fault injection metrics.

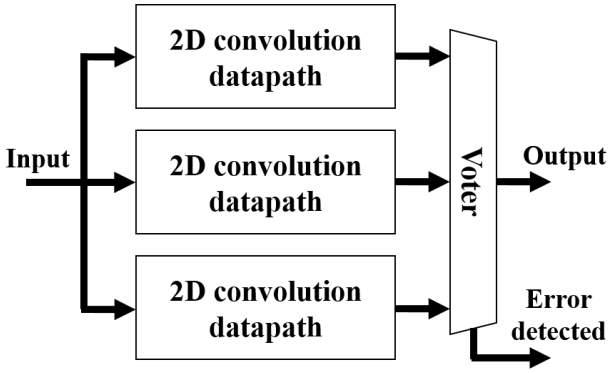



Figure 6.5: Two-dimensional convolution datapath, with Triple Modular Redundancy

Table 6.1: LEON3 CUT + associated fault injection infrastructure

| Module | LUTs | FFs | BRAMs | DCMs |
|---|---|---|---|---|
| CUT (LEON3) | 13,794 | 4,134 | 4 | - |
| Fault Generator | 499 | 359 | - | - |
| Sys. Input Contr. | 2,292 | 1,430 | - | - |
| Sys. Clock Contr. | - | - | - | 1 |
| Sys. Output Contr. | 1,251 | 315 | 4 | - |
| Fault Classifier | 403 | 92 | - | - |
| Main Control Unit | 1,094 | 67 | - | 1 |
| **Total** | **19,333 (12.8%)** | **6,397 (2.1%)** | **6 (1.4%)** | **2 (16.7%)** |

Table 6.2: 2D convolution datapath CUT + associated fault injection infrastructure

| Module | LUTs | FFs | BRAMs | DCMs |
|---|---|---|---|---|
| CUT (datapath) | 4,412 | 425 | - | - |
| Fault Generator | 499 | 359 | - | - |
| Sys. Input Contr. | 3,332 | 784 | - | - |
| Sys. Clock Contr. | - | - | - | 1 |
| Sys. Output Contr. | 166 | 24 | - | - |
| Fault Classifier | 403 | 92 | - | - |
| Main Control Unit | 1,094 | 67 | - | 1 |
| **Total** | **9,906 (6.6%)** | **1,751 (0.6%)** | **-** | **2 (16.7%)** |

Instead, in the second test case (i.e., 2D convolution datapath), the *System Input Controller* consists of an LFSR that pseudo-randomly generates 98 8-bit inputs for the 49 multipliers. The inputs generation process is repeated for 307,200 clock cycles in order to emulate the processing of an image composed of 640x480 pixels. The *System Output Controller* compresses the output values using a Multiple-Input Shift-Register (MISR), and compares the output signature with the golden one stored internally [21].

In the last considered test case, the *Error detected* signal of the TMR Voter (Figure 6.5) is con-

nected to the *Fault Classifier* to notify if a fault is detected by the TMR mechanism.

The *System Clock Controller* and the *Fault Generator* are the same among the three case studies. The *System Clock Controller* is composed of a single Digital Clock Manager (DCM) [240], necessary to synthesize the clock for the CUT. The *Fault Generator* consists of: a memory controller, that reads from the external memory a look-up table storing the positions of each essential bit in the bitstream, two LFSRs, that pseudo-randomly select the injection time and the essential bit position to flip, and a DPR controller which writes the CUT bitstreams in the configuration memory through the *Internal Configuration Access Port* (ICAP) of the FPGA device. The ICAP is driven by the *Fault Generator* using the maximum possible clock frequency (i.e., 100 MHz), providing a reconfiguration bandwidth of 3.2 Gbps [238].

Table 6.4 shows the partial bitstream sizes ($BS$) associated with the CUTs, the percentage of essential bits $\%EB$, the time needed to run the application ($T_{run}$), and the total injection time ($T_{inj}$), performing a number of runs equal to the number of extracted essential bits.

In general, the total injection time can be estimated as:

$$T_{inj} = \#EB \cdot (T_{run} + T_{DPR}) \tag{6.1}$$

$T_{DPR}$ is the sum of two contributions: the first is the time needed to configure, with the faulty bitstream, the reconfigurable partition in which the CUT is implemented, while the second represents the time needed to restore it, with the golden bitstream, at the end of each run. Since the

Table 6.3: 2D convolution datapath with TMR CUT + associated fault injection infrastructure

| **Module** | **LUTs** | **FFs** | **BRAMs** | **DCMs** |
|---|---|---|---|---|
| CUT (datapath + TMR) | 13,339 | 1,275 | - | - |
| Fault Generator | 499 | 359 | - | - |
| Sys. Input Contr. | 3,332 | 784 | - | - |
| Sys. Clock Contr. | - | - | - | 1 |
| Sys. Output Contr. | 166 | 24 | - | - |
| Fault Classifier | 436 | 118 | - | - |
| Main Control Unit | 1,094 | 67 | - | 1 |
| **Total** | **18,866 (12.5%)** | **2,627 (0.9%)** | **-** | **2 (16.7%)** |

Table 6.4: CUTs Bitstream size, percentage of *Essential Bits*, application execution time, and total injection time

| **CUT** | $BS[KB]$ | $\%EB$ | $T_{run}[ms]$ | $T_{inj}[s]$ |
|---|---|---|---|---|
| L3 Susan | 755.6 | 16.2 | 37.91 | 41,415 |
| L3 CRC32 | 755.6 | 16.2 | 20.94 | 24,552 |
| L3 IFFT | 755.6 | 16.2 | 395.65 | 395,514 |
| 2D conv. | 170,9 | 13.6 | 6.14 | 4,573 |
| 2D conv TMR | 478,4 | 13.6 | 6.14 | 4,573 |

faulty and the golden CUT bitstreams have the same size, $T_{DPR}$ can be computed as:

$$T_{DPR} = 2 \cdot \frac{BS}{f_{ICAP}} \tag{6.2}$$

where $BS$ is the bitstream size, and $f_{ICAP}$ represents the ICAP operating frequency (i.e., 100 MHz in our experiments).

It is worth mentioning that in all the considered test cases, the percentage of essential bits associated with the static part of the design (i.e., the fault injection infrastructure and the I/O interface of the CUT) in the CUT bitstream is less than the 0.7%. As explained in Section 6.2, these bits are not included in the fault injection process. This contribution can be treated as an error on the final computed metrics, since the fault injection infrastructure will not be present in the final circuit implementation. However, this error can be considered negligible if this kind of evaluation is made in the early design phases, where a very accurate measure is not required.

Table 6.5 shows the *Fault Classifier* results in terms of percentages of faults that caused an observalbe error (*Critical*), and those that have not caused any error (*Non-Critical*). The number of *Equivalent Injected Faults* (EIF) and the percentages are computed taking into account also the number of non-essential bits. Each non-essential bit composing the bitstream can be flipped without any effect on the circuit functionality, thus a fault in those bits can be considered as *Non-Critical*.

As can be seen from Table 6.5, in the third test case (i.e., *LEON3* running IFFT) the percentage of faults that cause an error is higher than the percentage in the first two test cases, since the application uses the floating point unit, which represents about 40% of the CUT area. In the last test case the number of *Critical* faults is greatly reduced with respect to the fourth test case, since all the faults detected by the TMR mechanism that do not lead to a Voter output error are considered *Non-critical*.

The proposed methodology and infrastructure has been compared, in terms of fault injection execution time, with the integrated HWIFI methods presented in [116], [18], and [166]. The fault injection platforms presented in [18, 116, 166] require a fixed $T_{DPR}$ of about $10\mu s$, since they reconfigure a single *frame* of the configuration memory, instead of the overall CUT reconfigurable

Table 6.5: Fault Injection classification results

| CUT | EIF | % Critical | % Non-Critical |
|---|---|---|---|
| L3 Susan | 6.18 M | 9.8 | 90.2 |
| L3 CRC32 | 6.18 M | 7.3 | 92.7 |
| L3 IFFT | 6.18 M | 13.6 | 82.4 |
| 2D conv. | 1.39 M | 12.4 | 87.6 |
| 2D conv TMR | 1.39 M | 1.3 | 98.7 |

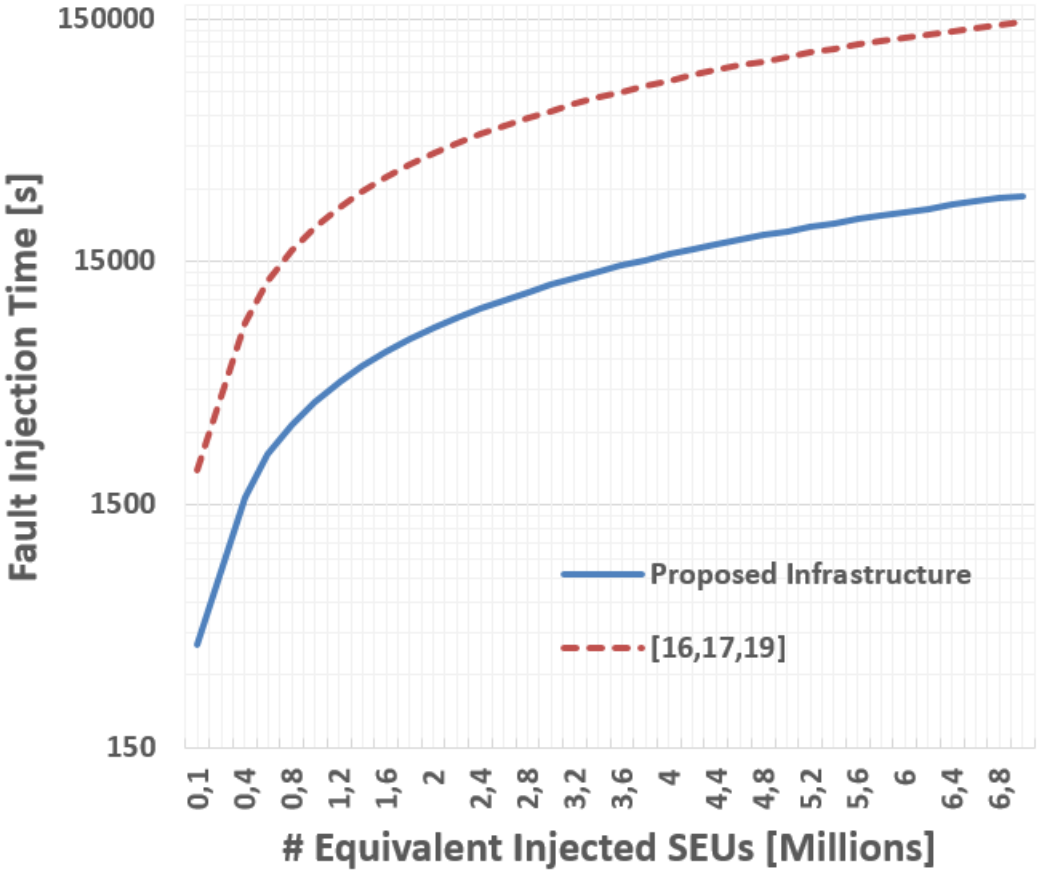



Figure 6.6: Fault injection time vs number of equivalent injected SEUs trends comparison, in the case of the LEON3 running CRC32

partition. Nonetheless, they incur in the problems described in Section 6.1.

Fig. 6.6 compares, in terms of total fault injection time ($T_{inj}$), the proposed method, with the ones proposed in [18, 116, 166], varying the number of equivalent injected faults (EIF). The reference case is the LEON3 processor running the CRC32 application.

As can be seen from Fig. 6.6, the proposed fault injection infrastructure extremely speed-up the fault injection process, especially when the number of SEUs to be injected is high. Moreover, it ensures the correct operations during the whole fault injection process, guaranteeing always reliable fault injection results, contrary to the platforms presented in [18, 116, 166].

# CONCLUSIONS

The presented research work have introduced and discussed methodologies and approaches for enhancing embedded real-time processing in mission-critical contexts.

After introducing the reasons why digital image processing is becoming a key technology to enable innovative and more efficient approaches in mission critical applications (see Chapter 2), the thesis discusses modern reconfigurable FPGA devices (see Chapter 3), focusing on Dynamic Partial Reconfiguration features and their increasing popularity. Both Chapter 2 and Chapter 3 highlight the issues that can affect embedded real-time image processing systems, in terms of unwanted effects on acquired digital images and dependability of the adopted technology.

Chapter 4 provides details on proposed approaches to enhance image processing algorithm robustness. In particular, self-adaptivity features have been introduced in selected hardware accelerators, in order to maintain constant, or improve, the quality of the algorithm results for a wide range of input conditions, that are not always fully predictable at design-time (e.g., noise level variations). This has been accomplished by measuring at runtime some characteristics of the input images, and then tuning the algorithm parameters based on such estimations. Dynamic reconfiguration features of modern reconfigurable FPGAs have been used in order to integrate run-time adaptivity features into the proposed hardware accelerators.

Chapter 5 discusses the proposed solutions for enhancing dynamic reconfiguration process dependability. In particular, it presents two alternative ways to safely enhance reconfiguration process dependability, and consequently provide safe adaptivity mechanisms. The former is essentially based on a set of rules to be applied at design-time, while the latter relies on the usage of a configurable hardware self-reconfiguration manager that must be instantiated within the target system.

Finally, Chapter 6 introduces the problem of evaluating systemâĂŹs robustness with respect to target fault models, proposing a methodology, along with an associated hardware platform, for emulating the effects of soft errors on modern dynamically reconfigurable FPGAs.

The entire work relied on a strong collaboration with the industry. This allowed to test and demonstrate the proposed approaches and solutions using realistic test cases and actual relevant technologies in the field. Thanks to these collaborations, the proposed solutions have improved the state-of-the-art, as demonstrated by the obtained results.

The final aim of the presented research work has been to demonstrate the suitability of the proposed solutions and approaches in mission-critical contexts.

APPENDIX A

# LIST OF SYMBOLS AND ACRONYMS

Due to the large number of symbols used in this thesis to support the description of covered material, this appendix provides the following list of abbreviations. This list is intended to help the reader identify the meaning of a given acronym in a fast and easy way.

| | |
|---|---|
| ADAS | Advanced Driver Assistance Systems |
| ADC | Analog-to-Digital Converter |
| ALM | Adaptive Logic Module |
| ASICs | Application Specific Integrated Circuits |
| ATE | Automated Test Equipment |
| BIST | Built-In Self-Test |
| CAD | Computer-Aided Design |
| CCD | Charge-Coupled Device |
| CLB | Configurable Logic Block |
| CRC | Cyclic Redundancy Code |
| COTS | Commercial-Off-the-Shelf |
| CPLDs | Complex Programmable Logic Devices |
| CUT | Circuit Under Test |

| | |
|---|---|
| DfT | Design-for-Testability |
| DFT | Discrete Fourier Transform |
| DMA | Direct Memory Access |
| DPR | Dynamic Partial Reconfiguration |
| DIMES | Descent Image Motion Estimation System |
| DSPs | Digital Signal Processors |
| ECC | Error Correcting Code |
| EDA | Electronic Design Automation |
| EDL | Entry, Descent and Landing |
| FEM | Features Extraction and Matching |
| FFs | Flip-Flops |
| FIFO | First-In First-Out |
| FPGAs | Field Programmable Gate Arrays |
| FSM | Finite State Machine |
| GNC | Guidance Navigation and Control |
| GPU | Graphics Processing Unit |
| HCPLDs | High Capacity Programmable Logic Devices |
| HDL | Hardware Description Language |
| HLS | High-Level Synthesis |
| HWIFI | Hardware-Implemented Fault Injection |
| IC | Integrated Circuit |
| ICAP | Internal Configuration Access Port |
| IT | Information Technology |
| LFSR | Linear Feedback Shift Register |
| LUTs | Look-Up Tables |
| MBU | Multiple Bit Upset |
| MOS | Metal-Oxide-Semiconductor |

| | |
|---|---|
| NRE | Non-Recurrent Engineering |
| OTF | Optical Transfer Function |
| PAL | Programmable Array Logic |
| PLAs | Programmable Logic Arrays |
| PLDs | Programmable Logic Devices |
| PROM | Programmable Read Only Memory |
| PSF | Point Spread Function |
| RTL | Register-Transfer Level |
| SBU | Single Bit Upset |
| SEB | Single Event Burnout |
| SECDED | Single Error Correction Double Error Detection |
| SEEs | Single Event Effects |
| SEFI | Single Event Functional Interrupt |
| SET | Single Event Transient |
| SEU | Single Event Upset |
| SEGR | Single Event Gate Rupture |
| SEL | Single Event Latch-Up |
| SERDES | Serializer/Deserializer |
| SIFT | Scale Invariant Feature Transform |
| SoC | System-on-Chip |
| SoPC | Systems-on-Programmable-Chip |
| SPLDs | Simple Programmable Logic Devices |
| SRAM | Static Random Access Memory |
| SURF | Speeded-Up Robust Features |
| SWIFI | Software-Implemented Fault Injection |
| TID | Total Ionizing Dose |
| TMR | Triple Modular Redundancy |

| | |
|---|---|
| TTM | Time-To-Market |
| UAVs | Unmanned Aerial Vehicles |
| VBN | Video-based Navigation |
| VHDL | Very High-Speed Integrated Circuits HDL |

# BIBLIOGRAPHY


[1] University of Oxford - Affine Covariant Regions Dataset. `www.robots.ox.ac.uk/~vgg/data/data-aff.html`.

[2] Hadi Parandeh Afshar. *Closing the Gap between FPGA and ASIC: Balancing Flexibility and Efficiency*. PhD thesis, École Polytechnique Fédérale de Lausanne, 2012.

[3] M. Alderighi, F. Casini, S. d'Angelo, M. Mancini, S. Pastore, and G.R. Sechi. Evaluation of single event upset mitigation schemes for SRAM-based FPGAs using the FLIPPER fault injection platform. In *Defect and Fault-Tolerance in VLSI Systems, 2007. DFT '07. 22nd IEEE International Symposium on*, pages 105–113, Sept 2007.

[4] Gregory Allen, Gary Swift, and Carl Carmichael. Virtex-4 vq static seu characterization summary. 2008.

[5] Altera Corporation. *Increasing Design Functionality with Partial and Dynamic Reconfiguration in 28-nm FPGAs (WP-01137-1.0)*, 2010.

[6] Altera Corporation. *Enabling High-Performance DSP Applications with Stratix V Variable-Precision DSP Blocks - WP-01131-1.1*, 2011.

[7] Altera Corporation. *A Safety Methodology for ADAS Designs in FPGAs - White paper (WP-01204-1.0))*, 2013.

[8] Altera Corporation. *A New FPGA Architecture and Leading-Edge FinFET Process Technology Promise to Meet Next-Generation System Requirements (WP-01220-1.1))*, 2015.

[9] Altera Corporation. *Stratix 10 Device Overview*, 2015.

[10] Maria E Angelopoulou, Christos-Savvas Bouganis, Peter YK Cheung, and George A Constantinides. Fpga-based real-time super-resolution on an adaptive image sensor. In *Reconfigurable Computing: Architectures, Tools and Applications*, pages 125–136. Springer, 2008.

[11] J. Arlat, M. Aguera, L. Amat, Y. Crouzet, J.-C. Fabre, J.-C. Laprie, E. Martins, and D. Powell. Fault injection for dependability validation: a methodology and some applications. *Software Engineering, IEEE Transactions on*, 16(2):166–182, Feb 1990.

[12] Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, et al. *Fundamental concepts of dependability*. University of Newcastle upon Tyne, Computing Science, 2001.

[13] M. Ayinala and K.K. Parhi. High-speed parallel architectures for linear feedback shift registers. *Signal Processing, IEEE Transactions on*, 59(9):4459–4469, Sept 2011.

[14] Donald G Bailey. *Design for embedded image processing on FPGAs*. John Wiley & Sons, 2011.

[15] C. Basile, S. Di Carlo, and A. Scionti. FPGA-based remote-code integrity verification of programs in distributed embedded systems. *Systems, Man, and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on*, 42(2):187–200, March 2012.

[16] N. Battezzati, S. Colazzo, M. Maffione, and L. Senepa. SURF algorithm in FPGA: A novel architecture for high demanding industrial applications. In *Proc. of 2012 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 161–162, 2012.

[17] Niccolò Battezzati, Luca Sterpone, and Massimo Violante. *Reconfigurable field programmable gate arrays for mission-critical applications*. Springer Science & Business Media, 2010.

[18] Niccolò Battezzati, Luca Sterpone, and Massimo Violante. *Reconfigurable field programmable gate arrays for mission-critical applications*. Springer Science & Business Media, 2010.

[19] Herbert Bay, Andreas Ess, Tinne Tuytelaars, and Luc Van Gool. SURF: Speeded up robust features. *Computer Vision and Image Understanding (CVIU)*, 110:346–359, 2008.

[20] P. Beaudet. Rotationally invariant image operators. In *Proc. of 4th International Joint Conference on Patter Recognition*, pages 579–583, 1978.

[21] Alfredo Benso and Paolo Prinetto. *Fault Injection Techniques and Tools for Embedded Systems Reliability Evaluation*. Springer, 2003.

[22] P. Bernardi, M.S. Reorda, L. Sterpone, and M. Violante. On the evaluation of SEU sensitiveness in SRAM-based FPGAs. In *On-Line Testing Symposium, 2004. IOLTS 2004. Proceedings. 10th IEEE International*, pages 115–120, July 2004.

[23] R. Bernstein. Digital image processing of earth observation sensor data. *IBM Journal of Research and Development*, 20(1):40–57, Jan 1976.

[24] Sheetal Bhandari, Fabio Cancare, Davide Basilio Bartolini, Matteo Carminati, Marco Domenico Santambrogio, and Donatella Sciuto. On the management of dynamic partial reconfiguration to speed-up intrinsic evolvable hardware systems. In *Proc. of the 6th HiPEAC Workshop on Reconf. Computing*, 2012.

[25] R. Bonamy, D. Chillet, S. Bilavarn, and O. Sentieys. Power consumption model for partial and dynamic reconfiguration. In *2012 International Conference on Reconfigurable Computing and FPGAs (ReConFig)*, pages 1–8, 2012.

[26] R. Bonamy, Hung-Manh Pham, Sebastien Pillement, and D. Chillet. Uparc: Ultra-fast power-aware reconfiguration controller. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2012*, pages 1373–1378, March 2012.

[27] D. Bouris, A. Nikitakis, and I. Papaefstathiou. Fast and efficient FPGA-based feature detection employing the SURF algorithm. In *18th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 3–10, 2010.

[28] Alan C Bovik. *Handbook of image and video processing*. Academic press, 2010.

[29] Ronald Newbold Bracewell and RN Bracewell. *The Fourier transform and its applications*, volume 31999. McGraw-Hill New York, 1986.

[30] Stephen Brown and Jonathan Rose. Architecture of FPGAs and CPLDs: A tutorial. *IEEE Design and Test of Computers*, 13(2):42–57, 1996.

[31] C. Cabani and W.J. MacLean. A proposed pipelined-architecture for FPGA-based affine-invariant feature detectors. In *Proc. of 2006 Computer Vision and Pattern Recognition Workshop (CVPRW)*, pages 121–126, 2006.

[32] Jian-Feng Cai, Hui Ji, Chaoqiang Liu, and Zuowei Shen. Blind motion deblurring from a single image using sparse approximation. In *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on*, pages 104–111. IEEE, 2009.

[33] Jian-Feng Cai, Hui Ji, Chaoqiang Liu, and Zuowei Shen. Framelet-based blind motion deblurring from a single image. *Image Processing, IEEE Transactions on*, 21(2):562–572, 2012.

[34] Patrizio Campisi and Karen Egiazarian. *Blind image deconvolution: theory and applications*. CRC press, 2007.

[35] Michael Cannon. Blind deconvolution of spatially invariant image blurs with phase. *Acoustics, Speech and Signal Processing, IEEE Transactions on*, 24(1):58–63, 1976.

[36] S.D. Carlo, G. Gambardella, M. Indaco, D. Rolfo, G. Tiotto, and P. Prinetto. An area-efficient 2-D convolution implementation on FPGA for space applications. In *Design and Test Workshop (IDT), 2011 IEEE 6th International*, pages 88–92, Dec 2011.

[37] M. Ceschia, M. Violante, M.S. Reorda, A. Paccagnella, P. Bernardi, M. Rebaudengo, D. Bortolato, M. Bellato, P. Zambolin, and A. Candelori. Identification and classification of single-event upsets in the configuration memory of SRAM-based FPGAs. *Nuclear Science, IEEE Transactions on*, 50(6):2088–2094, Dec 2003.

[38] J. Chen, L. Yuan, C.K. Tang, and L. Quan. Robust dual motion deblurring. In *Computer Vision and Pattern Recognition, 2008. CVPR 2008. IEEE Conference on*, pages 1–8. IEEE, 2008.

[39] Xiaogang Chen, Xiangjian He, Jie Yang, and Qiang Wu. An effective document image deblurring algorithm. In *Computer Vision and Pattern Recognition (CVPR), 2011 IEEE Conference on*, pages 369–376. IEEE, 2011.

[40] Yang Cheng, J. Goguen, A. Johnson, C. Leger, L. Matthies, M.S. Martin, and R. Willson. The mars exploration rovers descent image motion estimation system. *Intelligent Systems, IEEE*, 19(3):13–21, May 2004.

[41] Pong P Chu. *RTL hardware design using VHDL: coding for efficiency, portability, and scalability*. John Wiley & Sons, 2006.

[42] Pierluigi Civera, Luca Macchiarulo, Maurizio Rebaudengo, M Sonza Reorda, and Massimo Violante. Exploiting FPGA-based techniques for fault injection campaigns on VLSI circuits. In *Defect and Fault Tolerance in VLSI Systems, 2001. Proceedings. 2001 IEEE International Symposium on*, pages 250–258. IEEE, 2001.

[43] C. Claus, B. Zhang, W. Stechele, L. Braun, M. Hubner, and J. Becker. A multi-platform controller allowing for maximum dynamic partial reconfiguration throughput. In *Field Programmable Logic and Applications, 2008. FPL 2008. International Conference on*, Sept 2008.

[44] Christopher Claus, Johannes Zeppenfeld, Florian Müller, and Walter Stechele. Using partial-run-time reconfigurable hardware to accelerate video processing in driver assistance system. In *Proceedings of the conference on Design, automation and test in Europe*, pages 498–503. EDA Consortium, 2007.

[45] Cobham Gaisler. *GRLIB IP librry User's Manual*, 1.5.0 edition, January 2016.

[46] Altera Corporation. *Implementing FPGA Design with the OpenCL Standard (WP-01173-3.0)*, 2013.

[47] Philippe Coussy and Adam Morawiec. *High-level synthesis: from Algorithm to digital circuits*, volume 1. Springer, 2008.

[48] A. Cuoccio, P.R. Grassi, V. Rana, M.D. Santambrogio, and D. Sciuto. A generation flow for self-reconfiguration controllers customization. In *Electronic Design, Test and Applications, 2008. DELTA 2008. 4th IEEE International Symposium on*, pages 279–284, Jan 2008.

[49] J.J. Davis and P.Y.K. Cheung. Achieving low-overhead fault tolerance for parallel accelerators with dynamic partial reconfiguration. In *Field Programmable Logic and Applications (FPL), 2014 24th International Conference on*, pages 1–6, Sept 2014.

[50] G. Deng and L.W. Cahill. An adaptive gaussian filter for noise reduction and edge detection. In *Proc. of Nuclear Science Symposium and Medical Imaging Conference*, pages 1615 – 1619 vol.3, 1993.

[51] Adrien E Desjardins, Benjamin J Vakoc, Melissa J Suter, Seok-Hyun Yun, Guillermo J Tearney, and Brett E Bouma. Real-time fpga processing for high-speed optical frequency domain imaging. *Medical Imaging, IEEE Transactions on*, 28(9):1468–1472, 2009.

[52] S. Di Carlo, G. Gambardella, M. Indaco, D. Rolfo, G. Tiotto, and P. Prinetto. An area-efficient 2-D convolution implementation on FPGA for space applications. In *Proc. of 6th International Design and Test Workshop (IDT)*, pages 88 –92, 2011.

[53] S. Di Carlo, G. Gambardella, M. Indaco, D. Rolfo, G. Tiotto, and P. Prinetto. An area-efficient 2-D convolution implementation on FPGA for space applications. In *Proc. of 6th International Design and Test Workshop (IDT)*, pages 88 – 92, 2011.

[54] S. Di Carlo, G. Gambardella, P. Prinetto, D. Rolfo, P. Trotta, and P. Lanza. FEMIP: A high performance FPGA-based features extractor and matcher for space applications. In *23rd International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–4, 2013.

[55] S. Di Carlo, A. Miele, P. Prinetto, and A. Trapanese. Microprocessor fault-tolerance via on-the-fly partial reconfiguration. In *European Test Symposium (ETS), 2010. 15th IEEE*, pages 201–206, May 2010.

[56] S. Di Carlo, P. Prinetto, D. Rolfo, and P. Trotta. AIDI: An adaptive image denoising fpga-based ip-core for real-time applications. In *Adaptive Hardware and Systems (AHS), 2013 NASA/ESA Conference on*, pages 99–106, June 2013.

[57] S. Di Carlo, P. Prinetto, and A. Scionti. A FPGA-based reconfigurable software architecture for highly dependable systems. In *Asian Test Symposium, 2009. ATS '09.*, pages 125–130, nov. 2009.

[58] Stefano Di Carlo, Giulio Gambardella, Marco Indaco, Paolo Prinetto, Daniele Rolfo, and Pascal Trotta. Dependable dynamic partial reconfiguration with minimal area & time overheads on xilinx fpgas. In *Field Programmable Logic and Applications (FPL), 2013 23rd International Conference on*, pages 1–4. IEEE, 2013.

[59] Stefano Di Carlo, Giulio Gambardella, Piegiorgio Lanza, Paolo Prinetto, Daniele Rolfo, and Pascal Trotta. Safe: a self adaptive frame enhancer fpga-based ip-core for real-time space applications. In *Design and Test Symposium (IDT), 2013 8th International*, pages 1–6. IEEE, 2013.

[60] Stefano Di Carlo, Giulio Gambardella, Paolo Prinetto, Daniele Rolfo, and Pascal Trotta. Sa-femip: A self-adaptive features extractor and matcher ip-core based on partially reconfigurable fpgas for space applications. 2014.

[61] Stefano Di Carlo, Giulio Gambardella, Paolo Prinetto, Daniele Rolfo, Pascal Trotta, and Piegiorgio Lanza. Femip: A high performance fpga-based features extractor

& matcher for space applications. In *Field Programmable Logic and Applications (FPL), 2013 23rd International Conference on*, pages 1–4. IEEE, 2013.

[62] Stefano Di Carlo, Paolo Prinetto, Daniele Rolfo, Nicola Sansonne, and Pascal Trotta. A novel algorithm and hardware architecture for fast video-based shape reconstruction of space debris. *EURASIP Journal on Advances in Signal Processing*, 2014(1):1–19, 2014.

[63] Stefano Di Carlo, Paolo Prinetto, Daniele Rolfo, and Pascal Trotta. Aidi: An adaptive image denoising fpga-based ip-core for real-time applications. In *Adaptive Hardware and Systems (AHS), 2013 NASA/ESA Conference on*, pages 99–106. IEEE, 2013.

[64] Stefano Di Carlo, Paolo Prinetto, Daniele Rolfo, and Pascal Trotta. A fault injection methodology and infrastructure for fast single event upsets emulation on xilinx sram-based fpgas. In *Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT), 2014 IEEE International Symposium on*, pages 159–164. IEEE, 2014.

[65] Stefano Di Carlo, Paolo Prinetto, Pascal Trotta, and Jan Andersson. A portable open-source controller for safe dynamic partial reconfiguration on xilinx fpgas. In *Field Programmable Logic and Applications (FPL), 2015 25th International Conference on*, pages 1–4. IEEE, 2015.

[66] R. Dobai and L. Sekanina. Image filter evolution on the xilinx zynq platform. In *Adaptive Hardware and Systems (AHS), 2013 NASA/ESA Conference on*, pages 164–171, 2013.

[67] T. Drahonovsky, M. Rozkovec, and O. Novak. Relocation of reconfigurable modules on xilinx fpga. In *Design and Diagnostics of Electronic Circuits Systems (DDECS), 2013 IEEE 16th International Symposium on*, pages 175–180, April 2013.

[68] M. Dunstan, S. Parkes, and S. Mancuso. Visual navigation chip for planetary landers. In *Proc. of 2005 Conference on DAta Systems In Aerospace (DASIA)*, pages 1–7, 2005.

[69] M. Dunstan and M. Souyri. The FEIC development for NPAL project: A core image processing chip for smart landers navigation applications. In *MicroElectronics Presentation Days, ESA/ESTEC*, 2004.

[70] EADS Astrium. Navigation for planetary approach and landing - final report, [Accessed 28-July-2014].

[71] A. Ebrahim, T. Arslan, and X. Iturbe. On enhancing the reliability of internal configuration controllers in fpgas. In *Adaptive Hardware and Systems (AHS), 2014 NASA/ESA Conference on*, July 2014.

[72] A. Ebrahim, K. Benkrid, X. Iturbe, and Chuan Hong. A novel high-performance fault-tolerant icap controller. In *Adaptive Hardware and Systems (AHS), 2012 NASA/ESA Conference on*, June 2012.

[73] Michael Elad and Yacov Hel-Or. A fast super-resolution reconstruction algorithm for pure translational motion and common space-invariant blur. *Image Processing, IEEE Transactions on*, 10(8):1187–1193, 2001.

[74] F.A. Escobar, J. Tarrillo, Xin Chang, and C. Valderrama. Hardware managers with file system support for faster dynamic partial reconfiguration. In *Parallel and Distributed Processing with Applications (ISPA), 2014 IEEE International Symposium on*, pages 205–210, Aug 2014.

[75] Fairchild Imaging Corporation. *CCD424 1024 x 1024 Pixel Image Area Split Frame Transfer CCD Sensor*, 2002.

[76] Sina Farsiu, M Dirk Robinson, Michael Elad, and Peyman Milanfar. Fast and robust multiframe super resolution. *Image processing, IEEE Transactions on*, 13(10):1327–1344, 2004.

[77] Giuseppe Airo Farulla, Marco Indaco, Paolo Prinetto, Daniele Rolfo, and Pascal Trotta. Ablur: An fpga-based adaptive deblurring core for real-time applications. In *Adaptive Hardware and Systems (AHS), 2014 NASA/ESA Conference on*, pages 104–111. IEEE, 2014.

[78] J Fernández-Berni, R Carmona-Galán, F Pozas-Flores, Á Zarándy, and Á Rodríguez-Vázquez. Multi-resolution low-power gaussian filtering by reconfigurable focal-plane binning. In *SPIE Microtechnologies*, pages 806806–806806. International Society for Optics and Photonics, 2011.

[79] G Flandin, B Polle, B Frapard, P Vidal, C Philippe, and T Voirin. Vision based navigation for planetary exploration. In *32nd Annual AAS Rocky Mountain Guidance and Control Conference*, 2009.

[80] E.R. Fossum. Cmos image sensors: electronic camera-on-a-chip. *Electron Devices, IEEE Transactions on*, 44(10):1689–1698, Oct 1997.

[81] G. Foucard, P. Peronnard, and R. Velazco. Reliability limits of TMR implemented in a SRAM-based FPGA: Heavy ion measures vs. fault injection predictions. In *Test Workshop (LATW), 2010 11th Latin American*, pages 1–5, March 2010.

[82] Eiji Fujiwara. *Code design for dependable systems: theory and practical applications*. John Wiley & Sons, 2006.

[83] J. Gaisler. A portable and fault-tolerant microprocessor based on the SPARC v8 architecture. In *Dependable Systems and Networks (DSN), 2002. International Conference on*, pages 409–415, June 23–26, 2002.

[84] J. Gaisler. A portable and fault-tolerant microprocessor based on the SPARC v8 architecture. In *Dependable Systems and Networks, 2002. DSN 2002. Proceedings. International Conference on*, pages 409–415, 2002.

[85] Jiri Gaisler. A portable and fault-tolerant microprocessor based on the sparc v8 architecture. In *Dependable Systems and Networks, 2002. DSN 2002. Proceedings. International Conference on*, pages 409–415. IEEE, 2002.

[86] Gaisler Research AB. GR-CPCI-XC4V LEON PCI Virtex 4 development board - product sheet, [Accessed 28-July-2014].

[87] Joseph Gambarelli, Gérard Guérinel, Laurent Chevrot, and Mathieu Mattèi. *Computerized Axial Tomography: An Anatomic Atlas of Serial Sections of the Human Body AnatomyâĂŤRadiologyâĂŤScanner*. Springer Science & Business Media, 2012.

[88] Z. Ghaderi, S.G. Miremadi, H. Asadi, and M. Fazeli. HAFTA: Highly available fault-tolerant architecture to protect SRAM-based reconfigurable devices against multiple bit upsets. *Device and Materials Reliability, IEEE Transactions on*, 13(1):203–212, March 2013.

[89] E.A. Ghazaani, Z. Ghaderi, and S.G. Miremadi. A non-intrusive portable fault injection framework to assess reliability of FPGA-based designs. In *Field-Programmable Technology (FPT), 2013 International Conference on*, pages 398–401, Dec 2013.

[90] F. Giesemann, G. Paya-Vaya, H. Blume, M. Limmer, and W. Ritter. A comprehensive asic/fpga prototyping environment for exploring embedded processing systems for advanced driver assistance applications. In *Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIV), 2014 International Conference on*, pages 314–321, July 2014.

[91] Fei Gong, M. Vaidya, R. Kora, D. Harshbarger, B. Ulery, and W. Meyer. A fpga based prototype verification in automotive mixed signal integrated circuit development. In *Circuits and Systems (MWSCAS), 2013 IEEE 56th International Midwest Symposium on*, pages 1200–1203, Aug 2013.

[92] R.C. González and R.E. Woods. *Digital image processing 3rd edition*. Prentice Hall, 2007.

[93] R.C. González and R.E. Woods. *Digital Image Processing*. Pearson/Prentice Hall, 2008.

[94] Joseph W Goodman. *Introduction to Fourier optics*. Roberts and Company Publishers, 2005.

[95] S. Guillet, F. de Lamotte, N. Le Griguer, E. Rutten, J.-P. Diguet, and G. Gogniat. Modeling and synthesis of a dynamic and partial reconfiguration controller. In *Field Programmable Logic and Applications (FPL), 2012 22nd International Conference on*, pages 703–706, Aug 2012.

[96] M.R. Guthaus, J.S. Ringenberg, D. Ernst, T.M. Austin, T. Mudge, and R.B. Brown. MiBench: A free, commercially representative embedded benchmark suite. In *Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop on*, pages 3–14, Dec 2001.

[97] S. Habinc. Suitability of reprogrammable FPGAs in space applications - feasibility report. Technical report, Gaisler Research, September 2002.

[98] S.G. Hansen, D. Koch, and J. Torresen. High speed partial run-time reconfiguration using enhanced icap hard macro. In *Parallel and Distributed Processing Workshops*

*and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*, pages 174–180, May 2011.

[99] C. Harris and M. Stephens. A combined corner and edge detector. In *Proc. of the 4th Alvey Vision Conference*, pages 147 – 151, 1988.

[100] Scott Hauck and Andre DeHon. *Reconfigurable Computing: The Theory and Practice of FPGA-Based Computation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007.

[101] J. Heiner, N. Collins, and M. Wirthlin. Fault tolerant icap controller for high-reliable internal scrubbing. In *Aerospace Conference, 2008 IEEE*, pages 1–10, March 2008.

[102] I. Herrera-Alzu and M. Lopez-Vallejo. Design techniques for xilinx virtex fpga configuration memory scrubbers. *Nuclear Science, IEEE Transactions on*, 60(1):376–385, Feb 2013.

[103] John C Hoffman and Marios S Pattichis. A high-speed dynamic partial reconfiguration controller using direct memory access through a multiport memory controller and overclocking with active feedback. *International Journal of Reconfigurable Computing*, 2011, 2011.

[104] A. Hofmann, R. Wansch, R. Glein, and B. Kollmannthaler. An FPGA based onboard processor platform for space application. In *Adaptive Hardware and Systems (AHS), 2012 NASA/ESA Conference on*, pages 17–22, June 2012.

[105] M.M. Ibrahim, K. Asami, and Mengu Cho. Evaluation of SRAM-based FPGA performance by simulating SEU through fault injection. In *Recent Advances in Space Technologies (RAST), 2013 6th International Conference on*, pages 649–654, June 2013.

[106] Y. Ichinomiya, K. Takano, M. Amagasaki, M. Kuga, M. Iida, and T. Sueyoshi. Accelerated evaluation of SEU failure-in-time using frame-based partial reconfiguration. In *Field-Programmable Technology (FPT), 2012 International Conference on*, pages 220–223, Dec 2012.

[107] Cadence Design Systems Inc. *C-to-Silicon Compiler High-Level Synthesis: Automated high-level synthesis for design and verification*, 2011.

[108] Eric Jenn, Jean Arlat, Marcus Rimen, Joakim Ohlsson, and Johan Karlsson. Fault injection into VHDL models: the MEFISTO tool. In *Fault-Tolerant Computing, 1994. FTCS-24. Digest of Papers., Twenty-Fourth International Symposium on*, pages 66–75. IEEE, 1994.

[109] A. Johnson, R. Willson, J. Goguen, J. Alexander, and D. Meller. Field testing of the mars exploration rovers descent image motion estimation system. In *Robotics and Automation, 2005. ICRA 2005. Proceedings of the 2005 IEEE International Conference on*, pages 4463–4469, April 2005.

[110] Josh Broline and Nick van Vonno - Electronic Design. Hardening And Testing Semiconductor Devices In The Space Radiation Environment. `http://electronicdesign.com/analog/hardening-and-testing-semiconductor-devices-\space-radiation-environment`. Accessed: 2016-02-18.

[111] Neel Joshi, Sing Bing Kang, C Lawrence Zitnick, and Richard Szeliski. Image deblurring using inertial measurement sensors. *ACM Transactions on Graphics (TOG)*, 29(4):30, 2010.

[112] P. Kenterlis, N. Kranitis, A. Paschalis, D. Gizopoulos, and M. Psarakis. A low-cost SEU fault emulation platform for SRAM-based FPGAs. In *On-Line Testing Symposium, 2006. IOLTS 2006. 12th IEEE International*, pages 7 pp.–, 2006.

[113] P. Koopman. 32-bit Cyclic Redundancy Codes for internet applications. In *Dependable Systems and Networks (DSN), 2002. International Conference on*, pages 459–468, December 2002.

[114] P. Koopman. 32-bit cyclic redundancy codes for internet applications. In *Dependable Systems and Networks, 2002. DSN 2002. Proceedings. International Conference on*, pages 459–468, 2002.

[115] P. Koopman and T. Chakravarty. Cyclic Redundancy Code (CRC) polynomial selection for embedded networks. In *Dependable Systems and Networks (DNS), 2004. International Conference on*, pages 145–154, July 2004.

[116] U. Kretzschmar, A. Astarloa, J. Jimenez, M. Garay, and J. Del Ser. Compact and fast fault injection system for robustness measurements on SRAM-based FPGAs. *Industrial Electronics, IEEE Transactions on*, 61(5):2493–2503, May 2014.

[117] Dilip Krishnan and Rob Fergus. Fast image deconvolution using hyper-laplacian priors. In *Advances in Neural Information Processing Systems*, pages 1033–1041, 2009.

[118] Dilip Krishnan, Terence Tay, and Rob Fergus. Blind deconvolution using a normalized sparsity measure. In *Computer Vision and Pattern Recognition (CVPR), 2011 IEEE Conference on*, pages 233–240. IEEE, 2011.

[119] Ian Kuon, Russell Tessier, and Jonathan Rose. Fpga architecture: Survey and challenges. *Foundations and Trends in Electronic Design Automation*, 2(2):135–253, 2008.

[120] R.D. LaBelle and S.D. Garvey. Introduction to high performance ccd cameras. In *Instrumentation in Aerospace Simulation Facilities, 1995. ICIASF '95 Record., International Congress on*, pages 30/1–30/5, Jul 1995.

[121] P Lanza, A Martelli, Paolo Ernesto Prinetto, Daniele Rolfo, A Tramutola, and Pascal Trotta. Advanced image processing in space applications: the new trend to increase the success rate of exploration space missions.

[122] P Lanza, A Martelli, Paolo Ernesto Prinetto, Daniele Rolfo, A Tramutola, and Pascal Trotta. Fpga-based ip-cores library for advanced image processing in space applications. In *2013 International Space System Engineering Conference on Data Systems In Aerospace*.

[123] J.-C. Laprie. Dependable computing and fault tolerance : Concepts and terminology. In *Fault-Tolerant Computing, 1995, Highlights from Twenty-Five Years., Twenty-Fifth International Symposium on*, pages 2–, Jun 1995.

[124] Lattice Semiconductor. *MachXO2 Family Data Sheet (DS1035)*, 2015.

[125] U. Legat, A. Biasizzo, and F. Novak. Automated SEU fault emulation using partial FPGA reconfiguration. In *Design and Diagnostics of Electronic Circuits and Systems (DDECS), 2010 IEEE 13th International Symposium on*, pages 24–27, April 2010.

[126] Austin Lesea, Saar Drimer, Joseph J Fabula, Carl Carmichael, and Peter Alfke. The rosetta experiment: atmospheric soft error rate testing in differing technology FPGAs. *Device and Materials Reliability, IEEE Transactions on*, 5(3):317–328, 2005.

[127] Brian Leung and Seda Ogrenci Memik. Exploring super-resolution implementations across multiple platforms. *EURASIP Journal on Advances in Signal Processing*, 2013(1):116, 2013.

[128] Anat Levin, Yair Weiss, Fredo Durand, and William T Freeman. Understanding and evaluating blind deconvolution algorithms. In *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on*, pages 1964–1971. IEEE, 2009.

[129] C.C. Liebe. Star trackers for attitude determination. *Aerospace and Electronic Systems Magazine, IEEE*, 10(6):10–16, Jun 1995.

[130] Dave Litwiller. Ccd vs. cmos. *Photonics Spectra*, 35(1):154–158, 2001.

[131] Ming Liu, W. Kuehn, Zhonghai Lu, and A. Jantsch. Run-time partial reconfiguration speed investigation and architectural design space exploration. In *Field Programmable Logic and Applications, 2009. FPL 2009. International Conference on*, pages 498–502, Aug 2009.

[132] S. Liu, M. Boden, D.A. Girdhar, and J.L. Titus. Single-event burnout and avalanche characteristics of power dmosfets. *Nuclear Science, IEEE Transactions on*, 53(6):3379–3385, Dec 2006.

[133] Dominik Lorenz, Martin Barke, and Ulf Schlichtmann. Efficiently analyzing the impact of aging effects on large integrated circuits. *Microelectronics Reliability*, 52(8):1546 – 1552, 2012. {ICMAT} 2011 - Reliability and variability of semiconductor devices and {ICs}.

[134] E. Loupias, N. Sebe, S. Bres, and J.-M. Jolion. Wavelet-based salient points for image retrieval. In *Proceedings. 2000 International Conference on Image Processing*, volume 2, pages 518–521 vol.2, 2000.

[135] David G. Lowe. Object recognition from local scale-invariant features. In *Proceedings of the International Conference on Computer Vision-Volume 2 - Volume 2*, ICCV '99, Washington, DC, USA, 1999. IEEE Computer Society.

[136] LB Lucy. An iterative technique for the rectification of observed distributions. *The astronomical journal*, 79:745, 1974.

[137] A. Lustica. Ccd and cmos image sensors in new hd cameras. In *ELMAR, 2011 Proceedings*, pages 133–136, Sept 2011.

[138] Wassim Mansour and Raoul Velazco. SEU fault-injection in VHDL-based processors: A case study. *Journal of Electronic Testing*, 29(1):87–94, 2013.

[139] Clive Maxfield. *FPGAs: World Class Designs: World Class Designs*. Newnes, 2009.

[140] M.D. McFarlane. Digital pictures fifty years ago. *Proceedings of the IEEE*, 60(7):768–770, July 1972.

[141] Ronald W Mehler. *Digital Integrated Circuit Design Using Verilog and SystemVerilog*. Elsevier, 2014.

[142] Uwe Meyer-Baese and U Meyer-Baese. *Digital signal processing with field programmable gate arrays*, volume 65. Springer, 2007.

[143] Microsemi Corporation. *Axcelerator Family FPGAs Datasheet*, 2012.

[144] Microsemi Corporation. *FPGAs for Space Applications*, 2012.

[145] Microsemi Corporation. *IGLOO2 and SmartFusion2 65nm Commercial Flash FPGAs - Interim Summary of Radiation Test Results*, 2014.

[146] Microsemi Corporation. *IGLOO2 FPGA and SmartFusion2 SoC FPGA - DS0128 Datasheet*, 2015.

[147] Microsemi Corporation. *Radiation-Tolerant FPGAs*, 2015.

[148] Microsemi Corporation. *RTAX-S/SL and RTAX-DSP Radiation-Tolerant FPGAs Datasheet*, 2015.

[149] Microsemi Corporation. *RTG4 FPGA Fabric User Guide (UG0574)*, 2015.

[150] Lukas Miculka and Zdenek Kotasek. Generic partial dynamic reconfiguration controller for transient and permanent fault mitigation in fault tolerant systems implemented into fpga. In *Design and Diagnostics of Electronic Circuits Systems, 17th International Symposium on*, April 2014.

[151] JM Mogollon, H Guzman-Miranda, J Napoles, J Barrientos, and MA Aguirre. FTUNSHADES2: A novel platform for early evaluation of robustness against SEE. In *Radiation and Its Effects on Components and Systems (RADECS), 2011 12th European Conference on*, pages 169–174. IEEE, 2011.

[152] J. Mora, A. Gallego, A. Otero, E. de la Torre, and T. Riesgo. Noise-agnostic adaptive image filtering without training references on an evolvable hardware platform. In *2013 Conference on Design and Architectures for Signal and Image Processing (DASIP)*, pages 182–189, 2013.

[153] J. Mora, A. Gallego, A. Otero, B. Lopez, E. de la Torre, and T. Riesgo. A noise-agnostic self-adaptive image processing application based on evolvable hardware. In *2013 Conference on Design and Architectures for Signal and Image Processing (DASIP)*, pages 351–352, 2013.

[154] A.I. Mourikis, N. Trawny, S.I. Roumeliotis, A.E. Johnson, A. Ansar, and L. Matthies. Vision-aided inertial navigation for spacecraft entry, descent, and landing. *IEEE Transactions on Robotics*, 25(2):264–280, April 2009.

[155] P. Nangtin, P. Kumhom, and K. Chamnongthai. Video-based obstacle tracking for automatic train navigation. In *Proc. of 2005 International Symposium on Intelligent Signal Processing and Communication Systems (ISPACS)*, pages 21–24, 2005.

[156] NASA. NASA Curiosity Rover: Entry, Descent, and Landing, [Accessed 28-July-2014].

[157] NASA. Solar system exploration roadmap, [Accessed 28-July-2014].

[158] NASA Jet Propulsion Laboratory. Mission to Moon - Ranger 7. `http://www.jpl.nasa.gov/missions/ranger-7/`. Accessed: 2016-02-21.

[159] National Aeronautics and Space Administration. Final Minutes of Curiosity's Arrival at Mars. `http://www.nasa.gov/mission_pages/msl/multimedia/gallery/pia13282.html`. Accessed: 2016-02-22.

[160] National Aeronautics and Space Administration. Kepler 2 and K2 - Mission Overview. `http://www.nasa.gov/mission_pages/kepler/overview/index.html`. Accessed: 2016-02-22.

[161] National Aeronautics and Space Administration. NEO - NASA Earth Observation. `http://neo.sci.gsfc.nasa.gov/`. Accessed: 2016-02-22.

[162] National Aeronautics and Space Administration - Jet Propulsion Laboratory. Mars Exploration Rovers. `http://mars.nasa.gov/mer/overview/`. Accessed: 2016-02-23.

[163] National Aeronautics and Space Administration - Jet Propulsion Laboratory. Mars Science Laboratory: Curiosity Rover. http://mars.jpl.nasa.gov/msl/mission/overview/. Accessed: 2016-02-22.

[164] National Aeronautics and Space Administration - Jet Propulsion Laboratory. Mars Science Laboratory: Curiosity Rover - Mars Descent Imager (MARDI). http://mars.nasa.gov/msl/mission/instruments/cameras/mardi/. Accessed: 2016-02-22.

[165] National Aeronautics and Space Administration - Jet Propulsion Laboratory. Mars Science Laboratory: Eyes and Other Senses. http://mars.jpl.nasa.gov/msl/mission/rover/eyesandother/. Accessed: 2016-02-22.

[166] G.L. Nazar and L. Carro. Fast single-FPGA fault injection platform. In *Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT), 2012 IEEE International Symposium on*, pages 152–157, Oct 2012.

[167] Sergiu Nedevschi, Radu Danescu, Ciprian Pocol, and Marc Michael Meinecke. Stereo image processing for adas and pre-crash systems. In *proc of 5th International Workshop on Intelligent Transportation*, pages 55–60, 2008.

[168] Kyprianos Papadimitriou, Apostolos Dollas, and Scott Hauck. Performance of partial reconfiguration in fpga systems: A survey and a cost model. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 4(4):36, 2011.

[169] Gardana Pavlovic and A Murat Tekalp. Maximum likelihood parametric blur identification based on a continuous spatial domain model. *Image Processing, IEEE Transactions on*, 1(4):496–504, 1992.

[170] Edward Petersen. *Single event effects in aerospace*. John Wiley & Sons, 2011.

[171] Hung-Manh Pham, Van-Cuong Nguyen, and Trong-Tuan Nguyen. Ddr2/ddr3-based ultra-rapid reconfiguration controller. In *Communications and Electronics (ICCE), 2012 Fourth International Conference on*, pages 453–458, Aug 2012.

[172] Ioannis Pitas. *Digital image processing algorithms and applications*. Wiley.com, 2000.

[173] A. Putnam, A.M. Caulfield, E.S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G.P. Gopal, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J.-Y. Kim, S. Lanka, J. Larus, E. Peterson, S. Pope, A. Smith, J. Thong, P.Y. Xiao, and D. Burger. A reconfigurable fabric for accelerating large-scale datacenter services. In *Computer Architecture (ISCA), 2014 ACM/IEEE 41st International Symposium on*, pages 13–24, June 2014.

[174] Alex Rav-Acha and Shmuel Peleg. Two motion-blurred images are better than one. *Pattern Recognition Letters*, 26(3):311–317, 2005.

[175] F. Russo. A method for estimation and filtering of gaussian noise in images. *IEEE Transactions on Instrumentation and Measurement*, 52(4):1148 – 1154, 2003.

[176] Julio Sanchez and Maria P. Canton. *Space Image Processing*. CRC Press, Inc., Boca Raton, FL, USA, 1st edition, 1998.

[177] Ronald Donald Schrimpf and Dan M Fleetwood. *Radiation effects and soft errors in integrated circuits and electronic devices*, volume 34. World Scientific, 2004.

[178] Ronald Donald Schrimpf and Dan M Fleetwood. *Radiation effects and soft errors in integrated circuits and electronic devices*, volume 12. World Scientific, 2004.

[179] F. Schwiegelshohn, L. Gierke, and M. Hubner. Fpga based traffic sign detection for automotive camera systems. In *Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC), 2015 10th International Symposium on*, pages 1–6, June 2015.

[180] R. Scrofano, M.B. Gokhale, F. Trouw, and V.K. Prasanna. Accelerating molecular dynamics simulations with reconfigurable computers. *Parallel and Distributed Systems, IEEE Transactions on*, 19(6):764–778, June 2008.

[181] SEMICO research corporation. *How an FPGA Approach to Complex System Design Can Improve Profitability: Real Case Studies*, 2012.

[182] Debashis Sen and Sankar K Pal. Gradient histogram: Thresholding in a region of interest for edge detection. *Image and Vision Computing*, 28(4):677–695, 2010.

[183] O. Serres, V.K. Narayana, and T. El-Ghazawi. An architecture for reconfigurable multi-core explorations. In *Reconfigurable Computing and FPGAs (ReConFig), 2011. International Conference on*, pages 105–110, December 2011.

[184] Douglas Sheldon. Flash-based fpga nepp fy12 summary report.

[185] S. Shreejith and S.A. Fahmy. Extensible flexray communication controller for fpga-based automotive systems. *Vehicular Technology, IEEE Transactions on*, 64(2):453–465, Feb 2015.

[186] Shanker Shreejith, Kizhepatt Vipin, Suhaib A. Fahmy, and Martin Lukasiewycz. An approach for redundancy in flexray networks using fpga partial reconfiguration. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2013*, pages 721–724, March 2013.

[187] Felix Siegle, Tanya Vladimirova, Jørgen Ilstad, and Omar Emam. Mitigation of radiation effects in sram-based fpgas for space applications. *ACM Comput. Surv.*, 47(2):37:1–37:34, January 2015.

[188] S. M. Smith and J. M. Brady. SUSAN - a new approach to low level image processing. *International Journal of Computer Vision (IJCV)*, 23:45–78, 1995.

[189] Haengseon Son, Seonyoung Lee, and Kyungwon Min. Fpga implementation of uwb radar signal processing for automotive application. In *Wireless Technology Conference (EuWIT), 2010 European*, pages 49–52, Sept 2010.

[190] Man Mohan Sondhi. Image restoration: The removal of spatially invariant degradations. *Proceedings of the IEEE*, 60(7):842–853, 1972.

[191] A. Sreeramareddy, R. Kallam, A.R. Dasu, and A. Akoglu. Self-configurable architecture for reusable systems with accelerated relocation circuit (SCARS-ARC). In *Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010. IEEE International Symposium on*, pages 1–4, April 2010.

[192] L. Sterpone, D. Sabena, A. Ullah, M. Porrmann, J. Hagemeyer, and J. Ilstad. Dynamic neutron testing of dynamically reconfigurable processing modules architecture. In *Adaptive Hardware and Systems (AHS), 2013 NASA/ESA Conference on*, pages 184–188, June 2013.

[193] M. Straka, J. Kastil, and Z. Kotasek. Fault tolerant structure for SRAM-based FPGA via partial dynamic reconfiguration. In *Digital System Design: Architectures, Methods and Tools (DSD), 2010. 13th Euromicro Conference on*, pages 365–372, September. 2010.

[194] Martin Straka, Jan Kastil, Zdenek Kotasek, and Lukas Miculka. Fault tolerant system design and SEU injection based testing. *Microprocessors and Microsystems*, 37(2):155–173, 2013.

[195] Tomasz Szydzik, Gustavo M Callico, and Antonio Nunez. Efficient fpga implementation of a high-quality super-resolution algorithm with real-time performance. *Consumer Electronics, IEEE Transactions on*, 57(2):664–672, 2011.

[196] Shen-Chuan Tai and Shih-Ming Yang. A fast method for image noise estimation using laplacian operator and adaptive edge detection. In *Proc. of 3rd International Symposium on Communications, Control and Signal Processing (ISCCSP)*, pages 1077 – 1081, 2008.

[197] Yu-Wing Tai, Hao Du, Michael S Brown, and Stephen Lin. Correction of spatially varying image and video motion blur using a hybrid camera. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 32(6):1012–1028, 2010.

[198] Yu-Wing Tai, Ping Tan, and Michael S Brown. Richardson-lucy deblurring for scenes under a projective motion path. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 33(8):1603–1618, 2011.

[199] J. Tarrillo, F.A. Escobar, F. Lima Kastensmidt, and C. Valderrama. Dynamic partial reconfiguration manager. In *Circuits and Systems (LASCAS), 2014 IEEE 5th Latin American Symposium on*, pages 1–4, Feb 2014.

[200] Thales Alenia Space. Aerospace module speed and trajectory estimation - internal report. Technical report, 2012.

[201] The Planetary Society. Curiosity's shrinking landing ellipse. `http://www.planetary.org/blogs/emily-lakdawalla/2012/20120611-curiosity-landing-ellipse.html`. Accessed: 2016-02-22.

[202] Jing Tian and Li Chen. Image noise estimation using a variation-adaptive evolutionary approach. *IEEE Signal Processing Letters*, 19(7):395 – 398, 2012.

[203] Matteo Tomasi, Shrinivas Pundlik, and Gang Luo. Fpga–dsp co-processing for feature tracking in smart video sensors. *Journal of Real-Time Image Processing*, pages 1–17, 2014.

[204] Tinne Tuytelaars and Krystian Mikolajczyk. *Local Invariant Feature Detectors: A Survey*. Now Publishers Inc., 2008.

[205] C. Urbina-Ortega, G. Furano, G. Magistrati, K. Marinis, and A. Menicucci. Flash-based FPGAs in Space, design guidelines and trade-off for critical applications. In *Proc. of IEEE Conference on Radiation Effects on Components and Systems (RADECS)*, 2013.

[206] Isa Servan Uzun, Abbes Amira, and Ahmed Bouridane. Fpga implementations of fast fourier transforms for real-time signal and image processing. In *Vision, Image and Signal Processing, IEE Proceedings-*, volume 152, pages 283–296. IET, 2005.

[207] Isa Servan Uzun, Abbes Amira, and Ahmed Bouridane. Fpga implementations of fast fourier transforms for real-time signal and image processing. In *Vision, Image and Signal Processing, IEE Proceedings-*, volume 152, pages 283–296. IET, 2005.

[208] Wim Vanderbauwhede and Khaled Benkrid. *High-Performance Computing Using FPGAs*. Springer, 2013.

[209] Gorka Velez, Ainhoa Cortés, Marcos Nieto, Igone Vélez, and Oihana Otaegui. A reconfigurable embedded vision system for advanced driver assistance. *Journal of Real-Time Image Processing*, 10(4):725–739, 2014.

[210] C.Y. Villalpando, R.A Werner, J.M. Carson, G. Khanoyan, R.A Stern, and N. Trawny. A hybrid FPGA/Tilera compute element for autonomous hazard detection and navigation. In *Aerospace Conference, 2013 IEEE*, pages 1–9, 2013.

[211] K. Vipin and S.A. Fahmy. A high speed open source controller for fpga partial reconfiguration. In *Field-Programmable Technology (FPT), 2012 International Conference on*, pages 61–66, Dec 2012.

[212] K. Vipin and S.A. Fahmy. Zycap: Efficient partial reconfiguration management on the xilinx zynq. *Embedded Systems Letters, IEEE*, 6(3):41–44, Sept 2014.

[213] Wei Wang and Michael K Ng. On algorithms for automatic deblurring from a single image. *Journal of Computational Mathematics*, 30(1):80–100, 2012.

[214] O. Whyte, J. Sivic, A. Zisserman, and J. Ponce. Non-uniform deblurring for shaken images. *International Journal of Computer Vision*, 98(2):168–186, 2012.

[215] V. Winkler, J. Detlefsen, U. Siart, J. Buchler, and M. Wagner. Fpga-based signal processing of an automotive radar sensor. In *Radar Conference, 2004. EURAD. First European*, pages 245–248, Oct 2004.

[216] Loring Wirbel. *Xilinx SDAccel: A Unified Development Environment for Tomorrow's Data Center*, 2014.

[217] Francis C Wong. *Digital circuit testing: A guide to DFT and other techniques*. Elsevier, 2012.

[218] Xilinx. *ML403 Evaluation Platform*, v2.5 edition, May 2006.

[219] Xilinx Corp. *UltraScale Devices Maximize Design Integrity with Industry-Leading SEU Resilience and Mitigation*, wp462 (v1.0) edition, 2015.

[220] Xilinx Corporation. CPLD: what is a CPLD? `http://www.xilinx.com/cpld/`. Accessed: 2015-12-26.

[221] Xilinx Corporation. FPGA vs. ASIC: What is the Difference Between a FPGA and an ASIC? `http://www.xilinx.com/fpga/asic.htm`. Accessed: 2015-12-19.

[222] Xilinx Corporation. The Total Solution for Next-Generation Automotive Applications. `http://www.xilinx.com/applications/automotive.html`. Accessed: 2015-01-03.

[223] Xilinx Corporation. *MicroBlaze Processor Reference Guide*, January 2008.

[224] Xilinx Corporation. *PowerPC Processor Reference Guide*, January 2010.

[225] Xilinx Corporation. *Automotive Driver Assistance Systems: Using the Processing Power of FPGAs - White paper (WP399)*, 2011.

[226] Xilinx Corporation. *Continuing Experiments of Atmospheric Neutron Effects on Deep Submicron Integrated Circuits - White paper (WP286)*, 2011.

[227] Xilinx Corporation. *LogiCORE IP XPS HWICAP v5.01a (DS586)*, June 2011.

[228] Xilinx Corporation. *ML605 Hardware User Guide - UG534 (v1.8)*, 2012.

[229] Xilinx Corporation. *Partial Reconfiguration of Xilinx FPGAs Using ISE Design Suite - WP374*, 2012.

[230] Xilinx Corporation. *Partial Reconfiguration of Xilinx FPGAs Using ISE Design Suite (WP374 - v1.2)*, 2012.

[231] Xilinx Corporation. *Partial Reconfiguration User Guide*, ug702 (v12.3) edition, October 2012.

[232] Xilinx Corporation. *PRC/EPRC: Data Integrity and Security Controller for Partial Reconfiguration (XAPP887) v1.1*, June 2012.

[233] Xilinx Corporation. *7 Series FPGAs Configuration User Guide (UG470)*, 2013.

[234] Xilinx Corporation. *Command Line Tools User Guide (UG628)*, 2013.

[235] Xilinx Corporation. *LogiCORE IP AXI HWICAP v3.0 (PG134)*, December 2013.

[236] Xilinx Corporation. *LogiCORE IP Fast Fourier Transform v9.0 Product Guide - PG109*, 2013.

[237] Xilinx Corporation. *LogiCORE IP Soft Error Mitigation Controller v4.0 - Product Guide for Vivado Design Suite (PG036)*, 2013.

[238] Xilinx Corporation. *Partial Reconfiguration User Guide (UG702)*, 2013.

[239] Xilinx Corporation. *Partial Reconfiguration User Guide (UG702 - v14.5)*, 2013.

[240] Xilinx Corporation. *Virtex-6 FPGA Clocking Resources User Guide (UG362)*, 2013.

[241] Xilinx Corporation. *Vivado Design Suite User Guide: Partial Reconfiguration (UG909 - v2015.4)*, 2013.

[242] Xilinx Corporation. *7 Series FPGAs Configurable Logic Block User Guide (UG474)*, 2014.

[243] Xilinx Corporation. *7 Series FPGAs Memory Resources User Guide - UG473*, 2014.

[244] Xilinx Corporation. *7 Series FPGAs Overview - DS180*, 2014.

[245] Xilinx Corporation. *A Generation Ahead for Smarter Systems: 9 reasons why the Xilinx Zynq-7000 All Programmable SoC platform is the smartest solution*, 2014.

[246] Xilinx Corporation. *Device Reliability Report - Fourth Quarter 2013 (UG116)*, 2014.

[247] Xilinx Corporation. *Radiation-Hardened, Space-Grade Virtex-5QV Family Overview (DS192 - v1.4)*, 2014.

[248] Xilinx Corporation. *Space-Grade Virtex-4QV Family Overview (DS653 - v2.1)*, 2014.

[249] Xilinx Corporation. *Spartan-3AN FPGA Family Data Sheet (DS557)*, 2014.

[250] Xilinx Corporation. *Vivado Design Suite User Guide - Partial Reconfiguration v2014.4 (UG909)*, November 2014.

[251] Xilinx Corporation. *Vivado Design Suite User Guide: High-Level Synthesis - UG902 (v2015.4)*, 2015.

[252] Xilinx Corporation. *White Paper: UltraScale Devices Maximize Design Integrity with Industry-Leading SEU Resilience and Mitigation (WP462 - v1.0)*, 2015.

[253] Xilinx Corporation. Virtex-4 FPGA User Guide - UG070, [Online - accessed 28-July-2014].

[254] Lifan Yao, Hao Feng, Yiqun Zhu, Zhiguo Jiang, Danpei Zhao, and Wenquan Feng. An architecture of optimised SIFT feature detection for an FPGA implementation of an image matcher. In *International Conference on Field-Programmable Technology, 2009. FPT 2009.*, pages 30–37, 2009.

[255] Ian T Young, Jan J Gerbrands, and Lucas J Van Vliet. *Fundamentals of image processing*. Delft University of Technology Delft, The Netherlands, 1998.

[256] Jun Zhang, Weisong Liu, and Yirong Wu. Novel technique for vision-based UAV navigation. *IEEE Transactions on Aerospace and Electronic Systems*, 47:2731–2741, 2011.

[257] Feng Zhao, Qingming Huang, and Wen Gao. Image matching by normalized cross-correlation. In *Proc. of 2006 International Conference on Acoustics, Speech and Signal Processing(ICASSP)*, pages II:729–732, 2006.

[258] Zhen-Bing Zhao, Jin-Sha Yuan, Qiang Gao, and Ying-Hui Kong. Wavelet image denoising method based on noise standard deviation estimation. In *Proc. of International Conference on Wavelet Analysis and Pattern Recognition (ICWAPR)*, volume 4, pages 1910 – 1914, 2007.